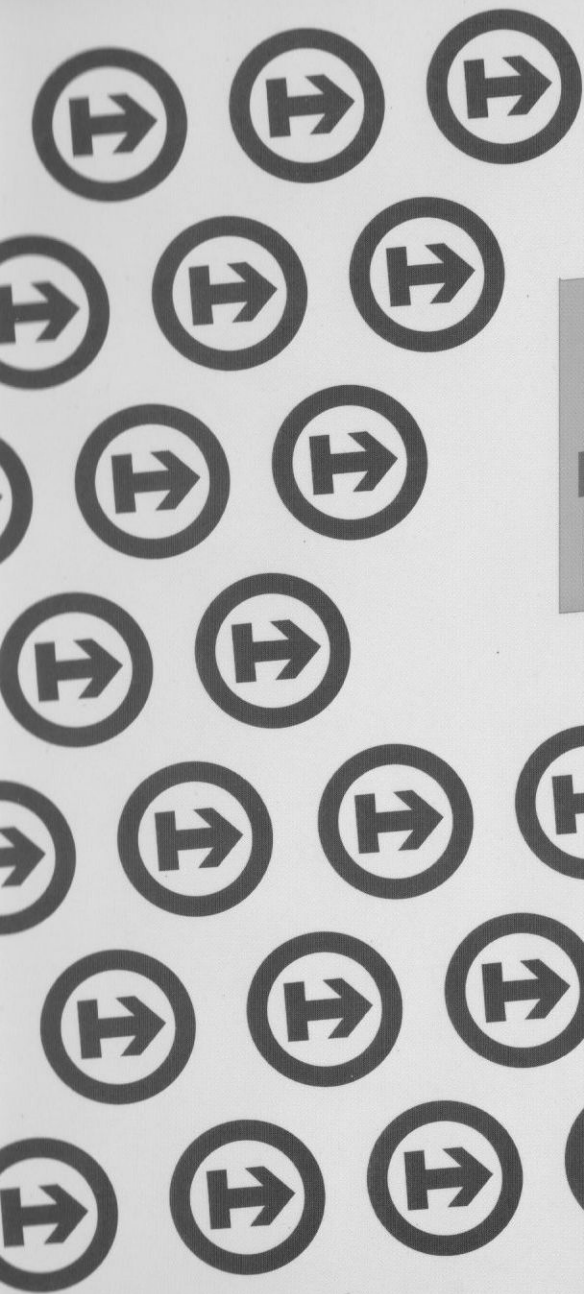




HighSpeed
Pascal

HiSoft
High Quality Software

***Technical
Reference***

Technical Reference

HighSpeed Pascal for the Amiga[®] By HiSoft and D-House

© Copyright 1991 HiSoft and D-House. All rights reserved.

Program
Designed and programmed by HiSoft's Clinton Fitt and D-House
Manual
Written by David Perkins, Alex Kierman and Keith Wilson

This guide and the HighSpeed Pascal program diskette contain proprietary information which is protected by copyright. No part of the software or the documentation may be reproduced, transmitted, stored in a retrieval system, translated into any language, or in any form without express prior written consent of the publisher.

HiSoft

High Quality Software

Published by HiSoft

The Old School, Greenfield, Bedford MK45 5DE UK

First Edition, January 1992 - ISBN 0 948517 51 4

Table of Contents

Chapter 1 The Language

Basic Symbols

Separators

Comments

Reserved Words

Identifiers

Numbers

Strings

Constant Declarations

Technical Reference

HighSpeed Pascal for the Amiga®

By HiSoft and D-House

© Copyright 1991 HiSoft and D-House. All rights reserved.

Program:

designed and programmed by HiSoft, Christen Fihl and D-House

Manual:

written by David Nutkins, Alex Kiernan and Keith Wilson

This guide and the HighSpeed Pascal program diskettes contain proprietary information which is protected by copyright. No part of the software or the documentation may be reproduced, transcribed, stored in a retrieval system, translated into any language or transmitted in any form without express prior written consent of the publisher and copyright holder(s).

HiSoft shall not be liable for errors contained in the software or the documentation or for incidental or consequential damages in connection with the furnishing, performance or use of the software or the documentation.

HiSoft reserves the right to revise the software and/or the documentation from time to time and to make changes in the content thereof without the obligation to notify any person of such changes.

Table of Contents

Chapter 1 The Language	1
Basic Symbols	1
Separators	2
Comments	2
Program lines	2
Reserved Words	3
Identifiers	3
Numbers	3
Strings	4
Constant Declarations	5
Compiler Directives	6
Types	6
Type declaration part	6
Type declaration	7
Simple Types/Ordinal Types	7
Integer	8
Boolean	8
Char	9
Enumerated	9
Subrange	9
Real	10
Structured Types	11
Array	11
Record	12
Set	14

File	14
String	15
Pointer	15
Variables	16
Variable declaration part	16
Variable declaration	16
Variables in different places	16
Use of Variables	17
Qualifiers	17
Arrays and String indexes	17
Records	18
Pointers	18
Variable Typecast	18
Typed Constants	20
Expressions	22
Arithmetic Operators	24
Statements	30
Simple Statements	30
Structured Statements	31
Procedure and Functions	37
Programs, Units and Procedures	43
Program Syntax	44
Unit Syntax	44
Input and Output	46
Typed and Untyped Files	49
Text Files	51
Error handling	54

Chapter 2 Language Reference	55
Chapter 3 The Supplied Units	217
The DOS Unit	217
Process-Handling Procedure	220
Environment functions	220
Directory handling procedures	221
Date and Time procedures	221
File Handling procedures and functions	222
Miscellaneous function	222
The System unit	223
The Crt unit	226
The Graph unit	228
Graph unit error codes	229
Appendix A AmigaDOS Errors	241
Appendix B Error Messages	245
Compiler Errors	245
Runtime Errors	253
Appendix C Inside HighSpeed Pascal	257
Memory Layout	257
Traps and Exceptions	258
Exit Procedures	258
Heap Manager	259

Data formats	259
Integer Types	259
Char Types	260
Enumerated Types	260
Boolean Types	260
Real Types	260
Pointer Types	261
Set Types	261
Array Types	262
Record Types	262
File Types	262
Calling Conventions	263
Value Parameters	265
Variable Parameters	266
Function Results	266
Using Assembly Language	266
Inline assembler	266
ASSEMBLER procedures and functions	272
Using External Assembly files	273
Inline	275
Device Drivers	276
CLI vs Workbench	278
CLI Startup	278
Workbench Startup	278

Chapter 1

The Language

Basic Symbols

The smallest units of text in a Pascal program are called tokens. They are classified into special symbols, word symbols, numbers and character strings.

These are the basic building blocks:

Letter	A to Z, a to z and underscore _
Digit	The ten digits from 0 to 9.
HexDigit	The normal digits, A to F and a to f.
Blank	All control characters and the space character.
Special	+ - * / = < > () [] { } , . ; : ' ^ @ \$ # < > <= >= := .. (*) (. .)

The last line in the above table consists of nine symbols, each of which is a pair of characters.

The following symbols have the same meaning:

(* and {

*) and }

(. and (

.) and)

Separators

Separators can be a blank, a newline or a comment. In fact all control characters will work. Two word symbols must be separated by at least one separator. Otherwise the compiler will only see one symbol.

Comments

Comments can be inserted anywhere where you are allowed to insert a blank or newline. A comment is bounded by { and } or by (* and *). A comment can span several lines.

```
{ This is a comment }  
(* This is also a comment *)  
{ This is also a comment *} }  
{ But this comment never ends! *)
```

If the first character right after the opening { or (* is a \$ character, then the comment is compiler directive.

Program lines

Each program line can be 127 characters long, the rest of the line is simply discarded. This can cause confusion as you will not necessarily get a compilation error. One example is starting a comment in position 1 but ending it in position 135 of the line. This comment will last forever, or until the end of another comment, thus causing part of your program to be ignored.

Reserved Words

The following words are reserved in HighSpeed Pascal, and cannot be redefined.

WordSymbol = and, array, begin, case, const, div, do, downto, else, end, external, file, for, forward, function, goto, if, implementation, in, inline, interface, label, mod, nil, not, of, or, packed, procedure, program, record, repeat, set, shl, shr, string, then, to, type, unit, until, uses, var, while, with, xor.

Identifiers

Identifiers denote labels, constants, variables, procedure, functions, units, programs and fields in records. An identifier consists of a letter symbol followed by letter and digit symbols.

Identifier =
Letter { Letter | Digit }

Numbers

The compiler is made for number crunching, so we need numbers too. Some examples:

7	9	13	
\$12	\$AA55	\$000F	
1.2	3.1415927	1.23e17	1e-9

Numbers used for integer types consist of only digits. These can also be written using hex notation, by using the \$ prefix.

Reals are formed using the normal engineering notation format. 1.23e4 is the same as 12300.00 or 12300.

UnsignedNumber =
UnsignedInteger | UnsignedReal.

UnsignedInteger =
DigitSequence | # HexDigitSequence.


```
UnsignedReal =  
    UnsignedInteger . DigitSequence [e ScaleFactor] |  
    UnsignedInteger e ScaleFactor.
```

```
ScaleFactor = [ Sign ] UnsignedInteger.
```

```
Sign = + | -
```

```
DigitSequence =  
    Digit { Digit }.
```

```
HexDigitSequence=  
    HexDigit { HexDigit }
```

```
SignedNumber =  
    SignedInteger | SignedReal.
```

```
SignedInteger =  
    [ Sign ] UnsignedInteger.
```

```
SignedReal =  
    [ Sign ] UnsignedReal.
```

Strings

A character string is a sequence of characters enclosed in single quotation marks. A quotation mark can itself be included by entering it twice.

A string can have a length of zero, one or more, it is always compatible with variables of type `string`. A string variable of length one is also compatible with variables of type `char`. With a length of `N` it is compatible with packed array `[1..N]` of `char`.

Like Turbo Pascal, HighSpeed Pascal can include control characters in the string as `#nn` or as `^C`. `#nn` enables you to write any ASCII character in the range 0 to 255. `^C` enables you to write normal control characters in the range from `Ctrl-'A'` to `Ctrl='Z'`. It is easier to write `^Z` than `#$1A` or `#26`.

```
CharacterString =  
    ' { StringElement } '
```

```
StringElement =  
    '' | AnyCharacterExceptApostrophe.
```

Some examples:

```
'Plain vanilla'  
'It's Ok'#13#10  
'  
'  
'a'  
'A'
```

The control characters entered into a string must be entered outside the apostrophes without using any spaces.

Constant Declarations

A constant declaration declares a constant identifier, and gives it a value.

```
ConstantDeclarationPart =  
    [ const ConstantDeclaration  
      { ConstantDeclaration } ].
```

```
ConstantDeclaration =  
    Identifier = Constant ;.
```

```
Constant =  
    [ Sign ] UnsignedNumber |  
    [ Sign ] ConstantIdentifier |  
    CharacterString.
```

```
ConstantIdentifier =  
    Identifier.
```

These constants are predefined in HighSpeed Pascal:

false, true	of type boolean
maxint, maxlongint	of type integer (longint)
pi	of type real (extended)

Compiler Directives

Source code control is achieved using compiler directives.

A compiler directives is a comment where the first character is a \$-sign. No spaces are allowed before the dollar.

```
{ $Define Debug}  
(* $Ifdef Debug *)  
{ $R-}  
{ $Not a compiler directive! }
```

Types

Every variable in Pascal has a type. The type of a variable determines the values that the variable can assume and the operations that can be performed on it.

Type declaration part

Type declarations typically appear in procedures and programs. The type declaration follows right after the **type** keyword. In HighSpeed Pascal there can be more than one **type** declaration block, and it can be mixed with constant, label, procedure, function and variable declarations.

```
TypeDeclarationPart =  
    type TypeDeclaration { TypeDeclaration }
```

Like Turbo Pascal, HighSpeed Pascal can include escape character in the string. For example, the character '\n' enables you to insert a new line character in the string. The character '\t' enables you to insert a tab character in the string. The character '\r' enables you to insert a carriage return character in the string. The character '\f' enables you to insert a form feed character in the string. The character '\a' enables you to insert an alert character in the string. The character '\b' enables you to insert a backspace character in the string. The character '\c' enables you to insert a control character in the string. The character '\d' enables you to insert a digit character in the string. The character '\e' enables you to insert an escape character in the string. The character '\f' enables you to insert a form feed character in the string. The character '\g' enables you to insert a graphic character in the string. The character '\h' enables you to insert a hexadecimal character in the string. The character '\i' enables you to insert an italic character in the string. The character '\j' enables you to insert a join character in the string. The character '\k' enables you to insert a keyboard character in the string. The character '\l' enables you to insert a lowercase character in the string. The character '\m' enables you to insert a macro character in the string. The character '\n' enables you to insert a new line character in the string. The character '\o' enables you to insert an octal character in the string. The character '\p' enables you to insert a punctuation character in the string. The character '\q' enables you to insert a quote character in the string. The character '\r' enables you to insert a carriage return character in the string. The character '\s' enables you to insert a space character in the string. The character '\t' enables you to insert a tab character in the string. The character '\u' enables you to insert an uppercase character in the string. The character '\v' enables you to insert a vertical tab character in the string. The character '\w' enables you to insert a word character in the string. The character '\x' enables you to insert a hexadecimal character in the string. The character '\y' enables you to insert a lowercase character in the string. The character '\z' enables you to insert a lowercase character in the string. The character '\A' enables you to insert an uppercase character in the string. The character '\B' enables you to insert a backspace character in the string. The character '\C' enables you to insert a control character in the string. The character '\D' enables you to insert a digit character in the string. The character '\E' enables you to insert an escape character in the string. The character '\F' enables you to insert a form feed character in the string. The character '\G' enables you to insert a graphic character in the string. The character '\H' enables you to insert a hexadecimal character in the string. The character '\I' enables you to insert an italic character in the string. The character '\J' enables you to insert a join character in the string. The character '\K' enables you to insert a keyboard character in the string. The character '\L' enables you to insert a lowercase character in the string. The character '\M' enables you to insert a macro character in the string. The character '\N' enables you to insert a new line character in the string. The character '\O' enables you to insert an octal character in the string. The character '\P' enables you to insert a punctuation character in the string. The character '\Q' enables you to insert a quote character in the string. The character '\R' enables you to insert a carriage return character in the string. The character '\S' enables you to insert a space character in the string. The character '\T' enables you to insert a tab character in the string. The character '\U' enables you to insert an uppercase character in the string. The character '\V' enables you to insert a vertical tab character in the string. The character '\W' enables you to insert a word character in the string. The character '\X' enables you to insert a hexadecimal character in the string. The character '\Y' enables you to insert a lowercase character in the string. The character '\Z' enables you to insert a lowercase character in the string.

```
CharacterString =  
    { StringElement }
```

```
StringElement =  
    AnyCharacterExceptApostrophe
```

Example:

```
Type
  NewInt      = Integer;
  BestInt     = NewInt;
  MyChar      = Char;
  Color       = (Red, Green, Blue);
  Byte        = 0..255;
  Weight      = 10..25;
  PRect       = ^Rect;
  Rect = Record
    x, y, w, h: Integer;
  end;
  Rects = Array [1..99] of Rect;
  Mixed = Array[1..4] of Record a,b: Integer end;
```

Type declaration

```
TypeDeclaration =
  Identifier = Type ;
```

```
Type =
  TypeIdentifier |
  SimpleType |
  StructuredType |
  StringType |
  PointerType
```

Simple Types/Ordinal Types

```
SimpleType =
  OrdinalType | RealType
```

A simple type defines an ordered set of values so that you can compare variables (of the same type) and see if one is greater than the other, for example.

```
OrdinalType =
  SubrangeType | EnumeratedType | OrdinalType
```

Ordinal types can be manipulated using these five routines:

Ord	returns the ordinality of a value
Succ	returns the value successive to the one given
Pred	returns the value preceding the one given
Inc	increments the value (default to next value)
Dec	decrements the value

The following relation is true:

$\text{Succ}(\text{Pred}(y)) = y$

The opposite effect to Ord can be obtained by using type casting. The boolean value `True=Boolean(1)` while `False=Boolean(0)` - no other boolean value is valid. Do not make constructs like this:

```
if Boolean(2)=true then ThisMightBeExecuted;
```

Integer, Boolean, Char, enumerated and subrange types are all described as *ordinal* types.

Integer

HighSpeed Pascal gives you five predefined integer types: `ShortInt`, `Byte`, `Integer`, `Word` and `LongInt`. They are defined:

```
ShortInt  = -128..127;  
Byte      = 0..255;  
Integer   = -32768..32767;      {-MaxInt-1..MaxInt}  
Word      = 0..65535;  
LongInt   = -2147483648..2147483647; {-MaxLongInt-1..MaxLongInt}
```

Variables of type `ShortInt` and `Byte` use one byte of memory. `Integers` and `Words` use 2 bytes, whereas `LongInts` use 4 bytes.

Boolean

The Boolean type defines two constants, `True` and `False`. They are defined as:

```
Boolean=(False,True);
```

Note that: `Ord(False)=0`, `Ord(True)=1` and thus `False<True`.

Char

The Char type gives 256 different character values. The character set used is the standard Amiga set. A problem that you may encounter concerns the definition of the upper 128 characters of the ASCII set, which are often used for accented national characters but are usually in different positions on different computers. If you intend porting your programs to other platforms be careful to research this beforehand.

The following relations hold on the Amiga (and on all machines that use the ASCII character set for the first 128 characters).

```
'0' < '1' < '2' < '3' < '4' < '5' < '6' < '7' < '8' < '9'
'A' < 'B' < ... 'Y' < 'Z'
'a' < 'b' < ... 'y' < 'z'
'9' < 'A' < 'Z' < 'a' < 'z'
```

Enumerated

```
EnumeratedType =
    ( IdentifierList ).
```

```
IdentifierList =
    Identifier { , Identifier }.
```

Examples of enumerated types:

```
Game = (Club, Diamond, Heart, Spade)
Weekday = (Monday, Tuesday, Wednesday, Thursday,
            Friday, Saturday, Sunday)
```

The first identifier has an ordinality of zero, the following an ordinality of one, and so on.

Subrange

A subrange, as the name suggests, gives you access to a limited range of the values in a host type. The subrange is specified by writing the lower and the upper limit required.

```
WorkDay = Monday..Friday;
ShortInt = -128..127;
```

Note that `Ord` of an enumerated type variable cannot be less than zero, while `Ord` of a subrange type can be any integer value.

Real

So much about ordinal types. The last `SimpleType` is `Real`. Real types, like normal ordinals, can be compared but they are not very useful for counting because there is no ordinality on reals.

All real types are predefined and cannot be extended by new user types.

Three kinds of reals are implemented: `Single`, `Double` and `Extended`.

Note: The type `Real` is the same as `Double`. This can be redefined if required simply by writing: `Type Real=Single;`

The type `Extended` is an internal format used when calculating or converting real values. It can be used as a normal type in your programs.

If used as `Type Real=Extended;` it will give:

- faster code because variables are not converted before use
- better precision with standard arithmetic
- not much better precision with the trigonometric functions

`Single` and `double` are IEEE compatible but the extended format is not. This may give problems if `Extended` values are used for disk files or other interface purposes.

`Single` reals can be moved around in memory because they only take four bytes and then can be moved with one machine code instruction. They still get converted when used in procedure calls and expressions.

Type	Range	Useful digits
Single	3.4e-38..3.4e38	7-8
Real=Double	1.7e-308..1.7e308	15-16
Extended	1.1e-4932..1.1e4932	18-19

Structured Types

Simple types can only hold one value for each variable. Structured types hold more than one value at a time. Moreover, a structured type may be *packed*. If the structured type is prefixed with *packed*, then the compiler will try to save storage. Set types are always packed.

The components of a structure may themselves be structured.

```
StructuredTypes =  
    StructuredTypeIdentifier |  
    [ packed ] UnpackedStructuredType.
```

```
UnpackedStructuredType =  
    ArrayType | RecordType | SetType | FileType.
```

```
StructuredTypeIdentifier =  
    TypeIdentifier.
```

Array

The array type has only one element type (the *component* type), but can contain a fixed number of them.

```
ArrayType =  
    array [ IndexType { , IndexType } ] of  
    ComponentType.
```

```
IndexType =  
    OrdinalType.
```

```
ComponentType =  
    Type.
```

An array can be multidimensional, in which case the declaration can be written in different ways:

```
Way1=array[boolean,7..9] of integer  
Way2=array[boolean] of array[7..9] of integer;
```

Both can be accessed (indexed) in either of these ways:

```
n:=Way1[true][8]; or n:=Way1[true,8];  
n:=Way2[true][8]; or n:=Way2[true,8];
```


Packed array[1..N] of char has special properties, not found in other array types: in particular such an array can be assigned to a string.

Note: When you write array[7..9] of integer; you create a new subrange type NoName=7..9; , it just doesn't have a name.

Record

Array types have one element type but could contain a fixed number of them.

Record types are able to contain a fixed number of components with different types.

```
RecordType =  
    record [ FieldList [ ; ] ] end.
```

```
FieldList =  
    FixedPart [ ; VariantPart ] | VariantPart.
```

```
FixedPart =  
    RecordSection { ; RecordSection }.
```

```
RecordSection =  
    IdentifierList : Type.
```

```
VariantPart =  
    case VariantSelector of Variant { ; Variant }.
```

```
Variant =  
    Constant { , Constant } ; ( FieldList ).
```

```
VariantSelector =  
    [ TagField : ] TagType.
```

```
TagType =  
    OrdinalTypeIdentifier.
```

```
TagField =  
    Identifier.
```

The IdentifierList names each element, as in a variable declaration.

A normal record without variant fields look like this:

```
DateRecord=
  record
    Year: Integer;
    Month: 1..12;
    Day: 1..31;
  end;
```

Whereas, with a variant field it looks like this:

```
Graphix=
  record
    x,y: Integer;
    case Obj: ObjType of
      Dot: ();
      Line: (x2,y2: Integer);
      Circle:(Diameter: Integer)
    end;
```

An example of its use:

```
var G: Graphics;
begin
  with G do begin
    x:=7; y:=8; Obj:=Line; x2:=13; y2:=45;
    DrawIt(G);
  end;
end;
```

One use of a variant record is for data conversion:

```
var CV: record
  case integer of {Does not use any space}
    0: (L: LongInt);
    1: (Hi,Lo: Integer);
  end;
begin
  CV.L:=$00110012;
  writeln(CV.Hi,' ',CV.Lo);
end;
```

Note that this example is inherently non-portable since it relies on the ordering of machine words. Whilst the same example will run under Turbo-Pascal on the PC it will give a different result!

Set

A set-type variable can contain zero or more elements of the base type.

It can only hold zero or one element of each value.

SetType = **set of** BaseType.

BaseType = OrdinalType.

In HighSpeed Pascal the ordinality of all elements of a set type must be within the range of 0..255. You cannot make a set type as: set of 0..999999.

Examples:

set of char

set of 10..20

Program Test;

Var

S: Set of 0..99;

begin

S:=[7,9,13,7]; {Three elements, 7, 9 and 13}

end.

File

The file type is a linear sequence of components that are all of same type. The component type can be any type as long as it does not include any file type elements.

The array type is similar to the file type, but is of fixed size. The file type can contain from zero to many elements.

The predefined type, *Text*, is a special file type in which the elements are lines of text. A variable of type *Text* is called a *textfile*.

Readln, eoln and other procedures work on textfiles.

FileType =
 file of ComponentType.

String

A string type value is a sequence of characters that has a dynamic length within a fixed maximum length. The length can range from zero (the empty string) to 255, a string of maximum length.

```
StringType =  
    string [ [ SizeAttribute ] ].
```

```
SizeAttribute =  
    UnsignedInteger.
```

The `SizeAttribute` specifies the maximum length allowed for the string variable, not the actual length of the string.

String type values can be compared using equal, greater-than etc.

Examples of comparison:

```
'Hello' = 'Hello'  
'Hello' < 'Hello World'  
'Hello' > ''  
'Hello' < 'hello'
```

The elements in a string can be accessed like the components of an array type.

There are special procedures and functions for manipulating strings and the operator `+` can be use for concatenating strings.

Pointer

A pointer type defines a set of values that point to variables (dynamic or not) of a specified type, the base type.

```
PointerType =  
    ^ BaseType.
```

```
BaseType =  
    OrdinalType.
```

The base type must be declared before the end of the current type definition block.

Values can be assigned to pointers with the `New` procedure, the `@` operator or the `Ptr` function.

Variables

Variable declaration part

The variable declaration lists all the variables used in a program, unit, procedure or function.

```
VariableDeclarationPart =  
    var VariableDeclaration { VariableDeclaration }.
```

Example:

```
var Counter,  
    M,N,O : Integer;  
    C1, C2 : Color;  
    BigSet : Set of Byte;  
    F1 : Text;  
    F2 : File of Rect;  
    FileMix : File of Mixed;
```

Variable declaration

A variable can only take values according to the type used for its declaration. A variable is undefined until it is assigned a value.

```
VariableDeclaration =  
    IdentifierList : Type {AbsoluteClause} ;.
```

Variables in different places

Global variables are those variables in programs and units, but outside of procedures and functions. Each global variable, other than arrays, can be no greater than 32Kb, but the total size of globals is not limited by the compiler.

Variables in a procedure and a function are created when the procedure/function is activated and removed right after the activation is terminated. Each procedure/function can only have 32Kb of local variables, but the total size of the stack can be as large as memory allows. Using too much stack space might stop the program with a runtime error.

Dynamic variables are created on the heap with **New** and removed again with **Dispose**. The heap can be as large as free memory allows.

Use of Variables

Using a variable is called *referencing* a variable.

```
VariableReference =  
    [ VariableIdentifier | VariableTypeCast ] {  
        Qualifier }
```

Qualifiers

```
Qualifier =  
    [ Index | FieldDesignator | ^ ]
```

A variable can be used just by writing its name thereby referencing the entire variable:

```
Rect  
Person
```

The variable can be followed by zero or more qualifiers, thereby referencing smaller and smaller parts of the variable:

```
Rect.Point1  
Rect.Point1.X  
Person[3].Name  
ZipCode[ Person[7].ZipIndex ]  
P^[N][8].R^^
```

Arrays and String indexes

Whenever the referenced part of a variable is an array, it can be qualified with an array index. Strings can be referenced like arrays.

```
Index =  
    [ Expression { , Expression } ].
```

The index used must be of an assignment compatible type with the corresponding index type used for the declaration.

Multiple indexes or multiple expressions within an index can be freely used:

```
AnArray[1,2,3] is the same as writing  
AnArray[1,2][3]      or as  
AnArray[1][2][3]
```

As mentioned, strings can be indexed as normal arrays. A string is always defined as `String[n]=array [0..n]` of `char`.

The first element is the actual length of the string. Obtaining the length of a string can then be done this way:

```
Length(Str) = Ord(Str[0])
```

Records

Whenever the referenced part of a variable is a record, it can be qualified with a field designator.

```
FieldDesignator =  
    . FieldIdentifier.
```

```
Person.Name  
Person.Name.First
```

Pointers

Whenever the referenced part of a variable is a pointer, it can be qualified by writing a pointer symbol after the variable.

```
Person.Father^  
Person.Next^  
P^
```

Variable Typecast

A variable reference can have its type changed by applying another type to it. This is done by writing the new type as a function taking the variable as argument, and returning the variable with the new type. The function name is the name of the new type required.


```
VariableTypeCast =
    TypeIdentifier ( VariableReference ).
```

type

```
TRect: record
    x,y: Integer;
end;
```

var

```
R: TRect;
L: Longint;
```

begin

```
L:=Longint(R);
TRect(L).y:=3;
Inc(Longint(R), $00020002);
end.
```

Absolute variables

An absolute clause specifies that a variable should be placed at an absolute address. This enables a variable to be mapped directly to hardware registers. You can also 'alias' variables so that one variable lies on top of another, thus removing the need for type-casting.

```
AbsoluteClause =
    absolute [UnsignedInteger | variable identifier]
```

Note that the use of this facility should be kept to the minimum; in general it is best to modify the hardware and vectors via the operating system. Using **absolute** for aliasing makes it very easy to write unmaintainable code.

Examples:

```
var aString: String[30];
    StringLength: byte absolute Str;
```

StringLength could then be used instead of Length(string).

```
var Vectors: array[0..255] of pointer absolute $0000;
```


Typed Constants

Typed constants are like normal variables but with an initial value. Normal constants are used for values that do not change as the program runs - their value cannot be changed because they don't have any memory reserved for them at run-time. Although typed constants are declared with the `Const` keyword they can be modified and they do take up space in the program's global area at runtime, even if they are declared local to a procedure or function. Thus, typed constants are similar to static variables in C or HiSoft BASIC.

The initialisation of a typed constant only occurs once when the program starts, even when the declaration is local to a procedure/function.

The most common use of Typed Constants is to initialise arrays that would otherwise require many separate assignment statements for each element.

```
TypeConstantDeclaration =  
    Identifier : Type = TypedConstant ;
```

```
TypedConstant =  
    Constant | ArrayConstant | RecordConstant |  
    SetConstant | SetConstant | NIL
```

Simple Constants

Constants of a simple type are assigned a value by writing the value straight after the equals sign:

```
Const EndChar: Char=#27; FirstPos: Integer=2;
```

String Constants

Strings are as easy to assign values to as simple types:

```
Const      Hello: String ='Hello World' #13#10;  
          ProgName:String[8]='MyDemo';
```

Array Constants

Array elements are written enclosed in parentheses and separated by commas.

```
ArrayConstant=  
( TypedConstant { , TypedConstant } )
```

Example:

```
Const YesNo:
```

```
Array[Boolean]of String[3]=('No','Yes');
```

Elements that are not included are initialised to binary zeros.

Example:

```
Const BigArray: Array[1..100] of Integer =  
(0,1,2,3,4,5); { the rest is zero }
```

Packed arrays of Char can be written as a simple string type constant:

```
Const Str: packed array [1..9] of char = 'abcdefghi';
```

Record Constants

Record fields are written using the field name and the assigned value separated by a colon. The fields themselves are separated by semi-colons:

```
RecordConstant=  
( Field ID : TypedConstant { ; FieldID :  
TypedConstant } )
```

Example:

```
Type TRect=  
Record x,y: integer; end;
```

```
Const Rect1: TRect=( x:0 ; y:0 );
```

Set Constants

Set elements are enclosed in square brackets and separated by commas. As with set expressions, set elements can be specified as a single value or as a range consisting of two constants separated by two periods:

```
SetConstant=  
  [ [ ElementConstant {, ElementConstant } ] ]
```

```
ElementConstant=  
  Constant [ .. Constant ]
```

Examples:

```
Const HexDigits: set of char =  
  ['0'..'9', 'a'..'f', 'A'..'F'];
```

```
YesNoChar: set of char= ['Y', 'y', 'N', 'n', #13];
```

Pointer Constants

The only legal pointer value that can be used is the value NIL. For example:

```
Const MyStructListHeader: Pointer =NIL;
```

Expressions

The productive part of a program is made up of expressions and statements. Expressions are made up of operators and operands. Some operators are unary; these only take one operand, for example, not true, while the others are binary and take two operands as in 2+3.

The operators have different precedence which tells which operator to evaluate first when more than one operator is applied at a time.

Parenthesized expressions are always evaluated first. If an operand is located between two operators of different precedence it is bound to the operator with the higher precedence. Sequences of operators of same precedence are executed from left to right.

Operator precedence:

Operators	Precedence	Class
@, not	1st	unary
*, /, div, mod, and, shl, shr	2nd	multiplying
,+ -, or, xor	3rd	adding
=, <>, <, <=,>, >=, in	4th	relational

Expression syntax:

Expression =

SimpleExpression [RelationalOperator
SimpleExpression].

SimpleExpression =

[Sign] Term { AddingOperator Term }

Term =

Factor { MultiplyingOperator Factor }

Factor =

UnsignedConstant |

Variable |

SetConstructor |

FunctionCall |

ValueTypeCast |

not Factor |

VariableReference |

@ VariableReference |

@ ProcedureIdentifier |

@ FunctionIdentifier |

(Expression).

UnsignedConstant =

UnsignedNumber |

CharacterString |

ConstantIdentifier | nil.

SetConstructor =

[[ElementDescription

{ , ElementDescription }]

].

ElementDescription =

Expression [.. Expression].

Arithmetic Operators

Arithmetic operators work on integer and real type operands and return integer and real results.

Unary arithmetic operations

Operator	Operation	Type of Operand	Type of result
+	none	integer/real	integer/real
-	negation	integer/real	integer/real

Binary arithmetic operations

Operator	Operation	Type of Operand	Type of result
+	addition	integer/real	integer/real
-	subtraction	integer/real	integer/real
*	multiplication	integer/real	integer/real
/	division	integer/real	real
div	integer division	integer	integer
mod	remainder	integer	integer

If one of the operands is `real` then the type of the result is always extended; if one of the operands is of type `longint` then the result is also `longint` otherwise the result is `integer`.

The rules used by `div` and `mod` are:

$$i \bmod j = i - (i \operatorname{div} j) * j$$

Examples:

+10 div +3 = +3	+10 mod +3 = +1
-10 div +3 = -3	-10 mod +3 = -1
+10 div -3 = -3	+10 mod -3 = +1
-10 div -3 = +3	-10 mod -3 = -1

Boolean Operators

Boolean operations

Operator	Operation
not	negates the boolean value (monadic)
and	takes the logical and
or	takes the logical or.
xor	takes the logical xor.

Boolean evaluation

Op1	Op2	and	or	xor
false	false	false	false	false
false	true	false	true	true
true	false	false	true	true
true	true	true	true	false

Logical Operators

The operators `not`, `and`, `or` and `xor` can also be used as bitwise operators. Bitwise operators work on all bits in the operand(s).

There are also two new operators `shl` and `shr`.

Operator	Operation
not	bitwise negation (monadic)
and	takes the logical and
or	bitwise or
xor	takes the logical xor
shl	shifts bitwise to the left
shr	shifts bitwise to the right

Examples:

`2 shl 3 = 16`

`$ffff xor $f = $ffff`

`not $a5f0 = $5a0f`

Set Operators

Three set operators exist

Operator	Operation
+	union
-	difference
*	intersection

The operands must be of compatible types, and the result is of type *set of a..b*, where *a* and *b* are the smallest and largest ordinal value that is a member of the result.

Set union ($a+b$) returns all members that are *either* in *a* or in *b* or in both.

Set difference ($a-b$) returns all members *in a but not in b*.

Set intersection ($a*b$) returns all members that are in *both a and b*.

Relational Operators

All relational operators return boolean values. They are used when comparing values against each other.

Comparing Ordinals

Operators applicable: $=, <>, <, <=, >, >=$.

Each ordinal operand must be of a compatible type. The result is the mathematical relation of their ordinalities.

Example:

Red < Green

Comparing Reals

Operators applicable: $=, <>, <, <=, >, >=$.

Each operand is first converted to *extended* before being tested; therefore take care when comparing reals to be equal and not-equal.

An example:

```
var
  ASingle: Single;
begin
  ASingle:=1/3;
  if ASingle=1/3 then NeverExecuted;
```

The first 1/3 is assigned as `single` precision while the next 1/3 is calculated as `extended` right away.

Comparing Strings

Operators applicable: `=`, `<>`, `<`, `<=`, `>`, `>=`.

All strings are compatible so all strings can be compared. A `char` variable can be treated as a string with actual length 1.

Examples of comparison:

```
'Hello' = 'Hello'
'Hello' < 'Hello World'
'Hello' > ''
'Hello' < 'hello'
'Hello' <= 'hello'
'Hello' <> 'hello'
```

Strings are compared according to the Amiga character set. This might be a problem when sorting strings with national characters included.

Comparing Packed Strings

Operators applicable: `=`, `<>`, `<`, `<=`, `>`, `>=`.

Both operands must have the same number of elements. Packed strings are compared like strings (with the same length).

Comparing Pointers

Operators applicable: `=`, `<>`.

Each pointer compared must be of a compatible type. Two pointers are only equal if they point to the same element.

Comparing Sets

Operators applicable: =, <>, <=, >=.

Both operands must be of compatible types. If *a* and *b* are sets, then:

a=*b* is true if every member of *a* is a member of *b* and visa versa.

a<>*b* is true if no member of *a* is a member of *b* and visa versa.

a<=*b* is true if every member in *a* is also a member of *b*.

a>=*b* is true if every member in *b* is also a member of *a*.

Testing Set Membership

Operators applicable: in.

The *in* operator tests to see if the ordinal type operand on the left is a member of the set operand on the right. If so it returns *true* else *false*.

The ordinal type on left side must be compatible with the base type of the right operand.

Example:

```
2 in [1..4] = true
2 in [7,9,13] = false
```

The @ Operator

The @ operator creates a pointer with a type like that of *nil*. The pointer can be assigned to any pointer variable. The operator works on variables, procedure and functions.

Do not use the @ operator if you don't have to. It is not a standard feature of Pascal; it was introduced by Turbo Pascal.

Because the type returned is compatible with all pointers, you do not get any type checking from the compiler, so you can easily (by accident) assign the address of a *char* to an *integer* pointer.

Function Calls

```
FunctionCall =  
    FunctionIdentifier [ ActualParameterList ].
```

```
ActualParameterList =  
    ( ActualParameter { , ActualParameter } ).
```

```
ActualParameter =  
    Expression | VariableReference.
```

A function call activates the function specified by the function identifier in the same way as a procedure call does. The procedure call does not return anything while a function call does return a value.

If the function declaration has a list of formal parameters the function call must also have a corresponding list of actual parameters.

Examples

```
Max(i,j)  
Random
```

Set Constructors

Set values are created by writing expressions within brackets.

```
SetConstructor =  
    [ [ MemberGroup ] ].
```

```
MemberGroup =  
    Expression { .. Expression }.
```

All members in the member group must be of compatible types. The result is of type set of a..b, where a and b are the smallest and largest ordinal value that is a member of the result.

Examples:

```
[ ]  
[ May, June ]  
[ '0'..'9', 'a'..'z' ]
```

Value Typecast

```
ValueTypeCast =  
    TypeIdentifier ( Expression ).
```

This changes the type of an expression from one type to another. The type returned is the one given, and the value is the one with the same ordinality. The expression must be of ordinal type or of pointer type.

Examples:

```
Char(65) {The same as chr(65)}  
Boolean(1) {Same as True}  
Longint(@Buffer)
```

Statements

Statements are said to be executable and can be *simple* or *structured*.

```
Statement =  
    { Label : } { SimpleStatement | StructuredStatement  
    }.  
Label =  
    DigitSequence | Identifier.
```

Simple Statements

The simple statement is a self contained statement.

```
SimpleStatement =  
    EmptyStatement |  
    AssignmentStatement |  
    ProcedureStatement |  
    GotoStatement.  
EmptyStatement = . (Nothing)
```

Structured Statements

The other main type of statements is a structured statement. It contains other statements which have to be executed:

- in sequence (compound statements),
- conditionally (if and case statements),
- repeatedly (repeat, while and for statements) or
- within an expanded scope (with statements).

```
StructuredStatement =  
  CompoundStatement |  
  IfStatement |  
  CaseStatement |  
  ForStatement |  
  RepeatStatement |  
  WhileStatement |  
  WithStatement. |  
  AsmStatement
```

Assignment

```
AssignmentStatement =  
  ( Variable | FunctionIdentifier ) := Expression.
```

A function identifier can only appear on the left side of a `:=` sign when assigning a return value to a function (within the function itself).

The expression must be assignment compatible with the type of the result variable or function identifier.

Examples:

```
n:=3;  
MyFunc:='Hello'  
Letters=['a'..'z', 'A'..'Z'];
```

Note: The `;` in last line is *not* part of the statement but a separator before the next (empty) statement.

Procedure Statements

ProcedureStatement =

ProcedureIdentifier [ActualParameterList] |

Write [WriteParameterList] |

Writeln [WriteParameterList].

A procedure call activates the procedure specified by the procedure identifier in the same way as a function call does. The procedure call does not return any value.

If the procedure declaration has a list of formal parameters the procedure call must also have a corresponding list of actual parameters.

Examples

SkipUntilEOF

Randomize

Writeln('Goodbye')

Write and Writeln are special because they can take a variable number of parameters, which is not allowed using normal procedure definitions.

Goto Statements

A label can prefix a statement which enables you to jump around in the program with `goto`. But try to avoid `gotos` and use the `repeat`, `while` and the other high-level statements instead.

GotoStatement =

goto Label.

Two rules must be remembered:

- The `goto` and the corresponding label must be in the same block.
- Do not jump into a structured statement. The `for` loop statement does some initial calculations that may not be skipped.

Compound Statements

Write `begin` and `end` around a sequence of statements, and you have formed a *compound* statement.

```
CompoundStatement =  
    begin StatementList end.
```

```
StatementList =  
    Statement { ; Statement }.
```

Note that it is legal to write as many semicolons as you like.

The last statement 'executed' could be empty, leaving only the `;` on the text line.

If Statements

The `if` statement is a two way switch. If the expression is true, the first way is taken, else the second.

```
IfStatement =  
    if expression then statement [ else statement ].
```

Note that it is *not* legal to put a semicolon before the `else` keyword; a semicolon is not a statement, but a separator.

Examples:

```
if OldEnough then WatchTV else GotoBed;  
if Bad then else GoodEnough;  
if not Bad then GoodEnough;
```

Case Statements

The `case` statement is a multi-way switch.

The expression is compared against the case constants one by one until a match is found. Then the corresponding statement is executed. If no match is made, either the `else` clause is executed or, if no `else` clause exists, nothing at all is executed.

The expression and all the case constants must be of same ordinal type.

```

CaseStatement =
    case expression of
    Case { ; Case }
    ElseClause
    { ; }
    end.

Case =
    ConstantElement { , ConstantElement } : Statement

ConstantElement =
    Constant [ .. Constant ].

ElseClause =
    else StatementList.

```

Examples:

```

Case NextSymbol of
  '+' : Expression:=n+Factor;
  '-' : Expression:=n-Factor;
Else
  writeln('Parsing error');
  halt(99);
end;

case Month of
  1,3,5,7,8,10,12: Days:=31;
  2: if Leap(Year) then Days:=29 else Days:=28;
  else Days:=30;
end;

```

For Statements

When you want to execute a statement (a compound statement etc.) a fixed number of times, you need the **for** repetitive statement.

```

ForStatement =
    for ControlVariable := InitialValue
    ( to | downto ) FinalValue do statement.

```

```

ControlVariable =
    VariableIdentifier.

```

```

InitialValue =
    Expression.

```

```

FinalValue =
    Expression.

```


The control variable must be a variable identifier which is either global or is defined local in the block containing the `for` loop. The variable must not be qualified, so do not use an array index, a record field or a pointer variable here.

The control variable assumes all values from initial value to final value including both. When using the `to` format the control variable increments by one for each time the loop is executed. When using `downto`, the control variable is decremented. The statement does not execute at all if the initial variable is greater than the final value when using the `to` format, and vice versa when using the `downto` format.

It is not legal to change the control variable within the loop. This is not checked by the compiler.

When the last iteration of the loop has finished the control variable is undefined. If you jump out of the loop using a `goto` statement, the control variable is still valid (containing the last value used).

Examples:

```
For n:=1 to 30000 do ; {Nothing}
```

```
For n:=1 to 9 do
  if Odd(n) then writeln(n,' is an odd number')
  else writeln(n,' is an even number');
```

Repeat Statements

The `repeat` statement executes some statements until an expression returns true. As the sequence of text lines shows, the expression is evaluated *after* the statements are executed, so the statements are always executed at least once.

```
RepeatStatement =
  repeat StatementList until Expression.
```

Example:

```
repeat write('*') until Keypressed;
```

While Statements

The while loop works almost as the repeat loop except for *when* the expression is evaluated. The while loop tests the expression *before* the statements are executed.

```
WhileStatement =  
    while Expression do Statement.
```

Examples:

```
while not eof(F) do begin  
    readln(F,S);  
    writeln('Line read: ',S)  
end;
```

With Statements

With statements open up a new scope and establish a reference to each of their associated record variables. While the statement is executing the reference does not change, even if you change the variables that made up the reference.

```
WithStatement =  
    with RecordVariableList do Statement.
```

```
RecordVariableList =  
    RecordVarReference { , RecordVarReference }.
```

Examples:

```
With SomeOther, Rect do begin  
    x:=7; y:=n;  
end;
```

This is equivalent to

```
With SomeOther do  
    With Rect do begin  
        x:=7; y:=n;  
    end;
```

This is equivalent to

```
Rect.x:=7; Rect.y:=n;
```

Another example:

```
P:=HeaderField;
while P<>Nil do
  with P^ do begin
    a:=AField; {Note a is the same as}
    P:=NextField;
    b:=AField; {b, because of with P^ do}
  end;
```

This is textually equivalent to writing

```
P:=HeaderField;
while P<>Nil do begin
  a:=P^.AField; {Now a is NOT equal}
  P:=NextField;
  b:=P^.AField; {to b!!!}
end;
```

but the program will behave in quite a different way.

Assembler statements

HighSpeed Pascal provides an extension to Pascal to provide in-line assembly language via the **ASM** statement. The general syntax is as follows:

```
AsmStatement=
  asm [Label:] AsmStmt {Separator AsmStmt} end
```

The ASM statement is discussed in detail in the *Inside HighSpeed Pascal* section.

Procedure and Functions

This section describes how to declare procedure and functions.

A procedure is activated by a **procedure** statement while a function is activated when used in an expression that contains its function name.

Procedures and functions are defined as:

```
ProcedureDeclaration =
  ProcedureHeading ; FuncProcBody ;.
```

```

FunctionDeclaration =FunctionHeading ; FuncProcBody ;.
FuncProcBody =
    Block |
    Forward |
    External |
    Assembler |
    Inline Constant { , Constant }.

ProcedureHeading =
    procedure Identifier [ FormalParameterList ].

FunctionHeading =
    function Identifier [ FormalParameterList ]
    : ResultType.

ResultType =
    TypeIdentifier | string.

```

If the procedure body is declared as a block, all the declaration is done right away. Alternatively, it may be declared as a forward declaration, an external declaration or an assembler declaration. Another way is to code a procedure as an inline declaration.

Examples:

```

procedure HexDump(Val,Format: Integer); forward;
Function Power10(N: Integer): Real;
procedure HexDump;
function Max(a,b: Integer): Integer; inline $xxxx,$xxxx;

```

Forward

A procedure that has the directive **forward** instead of the block is a forward declaration. Later, in the same declaration part, the procedure should be defined with its real block with a defining declaration. In a defining declaration, the parameter list (and a function's result type) is not repeated. In HighSpeed Pascal it is permissible to repeat the parameter list (and a function's result type), but this is *not* checked and *not* used for anything and is treated only as a comment.

Procedures named in the interface part of a unit work like forward declarations, but the keyword **forward** is not used.

Examples:

```
function Func(N: Integer): Integer; forward;

procedure Proc;
begin
  if Func(7)=3 then {};
end;

function Func;
begin
  Func:=N*2end;
```

External

A procedure that has the directive `external` instead of the block is an external declaration. The actual code block must be linked into the program by using a `{$L MyCode.obj}` directive somewhere in the program.

HighSpeed Pascal cannot check that the parameters are passed in the same way as the assembly code assumes.

Examples:

```
procedure BlockMove(var Src, Dst; Count: Integer);
external;
```

Assembler

Assembler procedures are used in conjunction with the inline assembler which is discussed in the *Inside HighSpeed Pascal* section.

Inline

A normal procedure call is done internally by making a JSR to the procedure. The `inline` directive permits you to replace this JSR with a sequence of instructions entered as constants after the `inline` directive.

Note that each procedure 'call' using `inline` inserts all constants listed, therefore keep them short.

HighSpeed Pascal cannot check that the parameters are passed in the same way as the inline code assumes.

A forward or interface procedure may not later be declared inline.

See *Inside HighSpeed Pascal* for further explanations.

Parameters

The declaration of a procedure (or function) can specify a *formal parameter list*. Each parameter in the list is local to the procedure in the same way as normal local parameters declared in a procedure. The only difference shows up in that it is initialised when the procedure is activated, and that it might reference another variable which is changed whenever the formal parameter is changed (a **var** parameter).

```
FormalParameterList =  
( ParameterDeclaration { ; ParameterDeclaration } ).
```

```
ParameterDeclaration =  
IdentifierList : ParameterType |  
var IdentifierList : ParameterType |  
var IdentifierList.
```

```
ParameterType = TypeIdentifier | string
```

There are three kinds of parameters: *value*, *variable* and *untyped variable* parameters.

Value Parameters

A value parameter is not preceded with **var**. It acts like a normal local variable, initialised when the procedure is initialised. The formal parameter can be changed without affecting the actual parameter used.

The actual parameter must be an expression that does not contain any file type elements. File types can only be passed as variable parameters.

If the parameter type is **string**, the formal parameter has a type of **String[255]**.

The actual parameter must be assignment compatible with the formal parameter.

Variable Parameters

A variable parameter is a parameter declaration preceded by **var** and followed by a type.

A formal variable parameter is located right on top of its actual parameter, so each time the formal parameter is changed, the actual parameter is also changed.

If the parameter type is string, the formal parameter has a type of **String[255]**, and the actual parameter must also have a length of 255. The type of the actual parameter and the type of the formal parameter must be identical.

Untyped Variable Parameters

An untyped variable parameter is a parameter declaration preceded by **var** but without any type following it.

The actual parameter passed can be of any type.

The formal parameter is typeless. It can only be used as a typeless pointer or by using variable type casting like in this example:

```
function Equal(var Src1, Src2; Length: Integer): Boolean;  
type
```

```
  Bytes= array [1..MaxInt] of ShortInt;
```

```
var
```

```
  n: Integer;
```

```
begin
```

```
  Equal:=False;
```

```
  for n:=1 to Size do
```

```
    if Bytes(Src1)[n]<>Bytes(Src2)[n] then Exit;
```

```
  Equal:=True
```

```
end;
```

```
var
```

```
  TestVar,TestVar2: array[1..99] of LongInt;
```

```
  B: Boolean;
```

```
  B:=Equal(TestVar,TestVar2,SizeOf(TestVar));
```

```
  B:=Equal(TestVar[10],TestVar2[20],SizeOf(LongInt));
```

```
  B:=Equal(TestVar[10],TestVar2[20],SizeOf(LongInt)*10);
```


Be careful when using untyped variables. It is almost like calling a function in the C language.

Procedure

A procedure is activated in a procedure statement like:

```
DumpSpaced(123).
```

Here is how a normal procedure might look:

```
procedure DumpSpaced(N: Longint);
var
  S: String[9];
begin
  Str(N,S);
  for N:=1 to Length(S) do
    write(S[N], ' ');
    writeln;
end.
```

This reuses N, without destroying the actual parameter. The procedure does not return anything.

A procedure can modify global variables, but this is discouraged, such procedures may have side effects.

Function

A function is activated when its name appears in a function call in an expression like: $R:=100*\text{Factorial}(10)+1$.

A normal function could look like this:

```
Function Factorial(N: Integer): Real;
begin
  if N<=1 then
    Factorial:=1
  else
    Factorial:=Factorial(N-1)*N
end;
```

A function must return a value. If it does not, you will get a garbage value returned.

A return value is assigned to the function when the function name appears on the left side of an assignment ($:=$). If the function name appears in any other place, it is treated as another (recursive) call to the function itself.

You cannot read the value back from the function name once it is assigned, you will have to keep a copy yourself.

You may assign a return value as many times as you like.

Programs, Units and Procedures

Blocks

Pascal programs consist of a number of *blocks*; a block can be a procedure, a function, a program or a unit. In each block, new identifiers can be declared, which are local to the block. The only exception is units, which declare identifiers for use by others.

Block = DeclarationPart StatementPart.

DeclarationPart = { LabelDeclarationPart |
ConstantDeclarationPart |
TypeDeclarationPart |
VariableDeclarationPart |
ProcedureDeclarationPart |
FunctionDeclarationPart }.

LabelDeclarationPart = **label** Label [,Label] ;

Label = Identifier | DigitSequence

ConstantDeclarationPart = **const** ConstantDeclaration
{ ConstantDeclaration }.

TypeDeclarationPart = **type** TypeDeclaration
{ TypeDeclaration }.

VariableDeclarationPart = **var** VariableDeclaration
{ VariableDeclaration }.

ProcedureDeclarationPart = ProcedureDeclaration.

FunctionDeclarationPart = FunctionDeclaration.

Program Syntax

A program looks a lot like a procedure, except for the first and last line. The heading is different and the last line reads `end`. instead of `end;`.

Program = [ProgramHeading] [UsesClause] Block .

ProgramHeading = **program** Identifier
[ProgramParameterList].

ProgramParameterList = (IdentifierList).

UsesClause = [**uses** IdentifierList].

The identifier after program names the program.

Unit Syntax

Units are used when making modular programs. Each unit has a name which must be the same as the name of the source file.

Unit= **unit** Identifier ;
InterfacePart
ImplementationPart
InitializationPart .

InterfacePart = **interface** UsesClause
{ ConstantDeclarationPart |
TypeDeclarationPart |
VariableDeclarationPart |
ProcedureHeadingPart |
FunctionHeadingPart }.

ProcedureHeadingPart = ProcedureHeading ; [
InlineDirective ;].

FunctionHeadingPart = FunctionHeading; [InlineDirective;]

ImplementationPart = **implementation**
{ LabelDeclarationPart |
ConstantDeclarationPart |
TypeDeclarationPart |
VariableDeclarationPart |
ProcedureDeclarationPart |
FunctionDeclarationPart }.

InitializationPart = **end** | StatementPart.

There is no label declaration in the interface part.

The implementation part must implement all the procedures and the functions defined in the interface part.

Everything not defined in the interface part is private to the unit.

The code entered in the initialisation part will be executed right before the first `begin` in the program module, executed in the same order as used.

Scope and Activations

The scope rules determine where an identifier can be used.

- An identifier cannot be used before it has been defined, reading the source text from top to bottom.
- An identifier cannot be used after the block in which it is defined ends.
- An identifier cannot be redefined in the same block.
- An identifier can be redefined in an enclosed block.
- An identifier can be reused in a record structure.

Scope for Units

Each unit used in a program or unit opens up a new scope. An identifier can be redefined in each of these units, and only the last seen will be used, except if it is referenced with a qualified name as `Unit3.N:=3;`.

Scope for Standard Identifiers

All identifiers defined by HighSpeed Pascal are defined in the System unit.

This unit is always the first one used, all identifiers can therefore be redefined.

Activations

When a procedure is called it is activated and an *activation record* is created for it.

An activation record contains much information including new space for each local variable each time the procedure is called.

Each time a procedure calls itself (recursively or mutually recursively) it gets new space for the locals. Each of the activation records does of course know where each of the variables for its own activation are.

Input and Output

All input and output routines in HighSpeed Pascal are described here. They are all predeclared; some of them are 'magic' in that they take a variable number of parameters, so they cannot be defined using normal Pascal programming.

There are three types of files in HighSpeed Pascal: text, typed and untyped files. They are declared as:

var

```
aTextFile:      Text;  
aTypedFile:     File of Something;  
anUntypedFile:  File;
```

All files used must have a name associated with them; the standard way of doing this is via the **assign** procedure. This name is what the file is called on the physical device where it is stored (normally a disk).

All files must be opened before use (giving them a name at the same time if you haven't already used the **assign** procedure). After use they *must* be closed again. This is particularly important with files that are open for writing since if you don't close such a file, it cannot be accessed or deleted until you reset the machine!

You can open an existing text file for input or for output append, to write generally to a text file you must create it, deleting any existing file first.

Other file types can be opened for input and output.

Text files are always accessed sequentially, you cannot seek around in them. Using other file types you can set the current read/write position by using the `seek` procedure.

In addition to the facilities provided by the operating system for using things like the screen and printer as files, HighSpeed Pascal also lets you write your own specialised device drivers.

After all calls to the input/output system an I/O-check routine is called to see if anything went wrong in the last routine. This can be disabled using the directive `{ $I - }` or by disabling the I/O checks in the Compiler Settings requester.

Routines for Input and Output

These are the routines that can be used for file I/O:

`Assign, Reset, Rewrite, Append, Close,`
`Seek, FilePos, FileSize,`
`Eof, Eoln, SeekEof, SeekEoln,`
`Read, Readln, Write, Writeln, Page, BlockRead, BlockWrite,`
`Rename, Erase,`
`IOResult, SetTextBuf.`

General control routines

These are the routines used for creating, erasing, renaming and opening files:

`Reset, Rewrite, Append, Close, Rename, Erase, IOResult.`

Assign(F, Title)

Associates the name `Title` with the file `F`. This name will be kept until another `Assign` is used on the file or `reset`, `rewrite` or `append` is called with two parameters.

Reset (F)

Open a file using the name previously used given to `assign`. The file must be read only if it is a text file. If the file is already opened it is rewound to the start of the file.

Reset (F, Title)

Open the file named *Title*. This is equivalent to

```
Assign(F,Title); Reset(F);
```

Rewrite(F)

Creates and opens a file with name associated with *assign*. Any previous file of the same is deleted. The file is write only if it is a text file.

Rewrite(F, Title)

Creates and opens the file named *Title*. This is equivalent to

```
Assign(F,Title); Rewrite(F);
```

Append(F)

Start appending on a text file that is already open. *Append* is like *rewrite* except that the writing starts at the end of any existing file.

Append(F, Title)

Start appending on a text file called *Title*. This is equivalent to

```
Assign(F,Title); Rewrite(F);
```

Close(F)

Flush unwritten data to the device and close the file. Ensure that you call this for every file.

Rename(OldName, NewName)

Rename a file from *OldName* to *NewName*. Both parameters are string types.

Erase(Title)

Delete the file from the disk.

IOResult: Integer

Return the result of last input or output operation. Zero is returned when no errors have occurred. **IOResult** is cleared by this call, so that subsequent calls will return 0 until another error happens.

Typed and Untyped Files

An untyped files is treated as a typed file with a type of: **file** of **ShortInt**.

These are the routines to use on typed and untyped files:

Seek, FilePos, FileSize, Eof, Read, Write, BlockRead, BlockWrite.

Seek(F, N: Longint)

Position the current read and write position. The first position is number zero.

FilePos(F): Longint

Return the current file position. **FilePos** returns zero after **reset** and **rewrite**.

FileSize(F): Longint

Return the number of elements in the file **F**.

Eof(F): Boolean

Returns false if there is still data to read after the current file position.

Read(F, v { ,vn })

Read one or more file components into one or more variables. The variable(s) **v** must be of the same type as the component type of **F**. The current file position is advanced by one position for each variable.

Write(F, v { ,vn })

Write one or more file components from one or more variables. The variable(s) *v* must be of same type as the component type of *F*. The current file position is advanced by one position for each variable.

BlockRead (F, Buffer, Count)

Read *Count* bytes from file *F* into *Buffer*. An *IOError* is returned if *Count* bytes could not be read.

BlockRead (F, Buffer, Count, ActualCount)

Read *Count* bytes from file *F* into *Buffer*. The actual count read is returned in *ActualCount*. No errors are generated.

BlockWrite (F, Buffer, Count)

Write *Count* bytes to file *F* from *Buffer*. An *IOError* occurs if *Count* bytes could not be written.

BlockWrite (F, Buffer, Count, ActualCount)

Write *Count* bytes to file *F* from *Buffer*. The actual count written is returned in *ActualCount*. No errors are generated.

BlockRead and *BlockWrite* can be used on both typed and untyped files.

An untyped file is treated as a typed file with a type of: *file of ShortInt*. The *ShortInt* size tells *reset*, *rewrite* and *seek* that the element size is one byte. *BlockRead* and *BlockWrite* take their sizes as byte counts. To read a *LongInt* from a file of *LongInt* you will have to use:

BlockRead(*F*,*B*,1*SizeOf(*LongInt*)).

Text Files

These are the routines for text file I/O:

Read, ReadLn, Write, WriteLn, Page, Eof, Eoln, SeekEof, SeekEoln, SetTextBuf.

For text file routines, you do not have to specify a file variable; if you do not, the file is then assumed to be either the standard file Input or Output.

Write and **WriteLn** take **Output** as their default file, while all the other procedures mentioned above take **Input** as their default.

Read (F, v { , Vn })

The variable **V** can be any one of these types: **Integer, Real, Char, String**.

When reading **Integers** or **Reals**, **HighSpeed Pascal** expects to read a sequence of digits and special symbols like **.**, **+**, **-**, **e**, or **E**. Any white spaces (tab, space and end-of-line) are skipped first.

When reading to a **Char** variable, one byte is read from the file. In standard Pascal the end-of-line symbol is read as a space, **chr(32)**, but in **HighSpeed Pascal**, you will read it as a new line, **chr(10)**.

Reading a string will read until the end of line is met. As many characters as the length of the string allows will then be put into the string. Reading a string will never read past a newline. If that required, then use **ReadLn**.

ReadLn (F, v { , Vn })

ReadLn does exactly the same as Read, just followed with a ReadLn(F).

```
ReadLn(F,v1,v2,v3);
```

is the same as:

```
Read(F,v1);
```

```
Read(F,v2);
```

```
Read(F,v3);
```

```
ReadLn(F);
```

Write (F, p { , Pn })

WriteLn (F, p { , Pn })

The variable P can be any one of these types: Integer, Real, Char, String, PackedString, Boolean.

```
WriteLn(F,p1,p2,p3);
```

is the same as:

```
Write(F,p1,p2,p3);
```

```
WriteLn(F);
```

which is the same as:

```
Write(F,p1);
```

```
Write(F,p2);
```

```
Write(F,p3);
```

```
WriteLn(F);
```

The write parameters p, P1, P2.. Pn have the form:

```
Expression [ [ : MinWidth ] : DecPlaces ] .
```

where Expression is the required output, and MinWidth and DecPlaces are integer expressions.

At least MinWidth positions are used when writing the expression. If the expression needs more space, more space is used. The default for MinWidth is 0.

DecPlaces specifies how many decimal digits a real number should have when converted to text; it can only be specified if **MinWidth** is also present.

Integer, **Char**, **String** and **PackedString** type variables are written left justified in a field with width **MinWidth**. If **MinWidth** is too small, no spaces are placed in front.

Boolean type variables are written by either writing the string **FALSE** or **TRUE**.

Real type variables are more complicated. They are first converted to text format using the **MinWidth** and **DecPlaces** value.

If **MinWidth** is omitted, it is assumed to be 10. If **DecPlaces** is omitted, the number is converted to a floating point decimal string, otherwise it is converted to a fixed point decimal string.

Floating point decimal string:

(| -) digit . decimals e (+ | -) exponent.

Fixed point decimal string:

[blanks] - digits . decimals.

Eof (F): Boolean

Returns false if there is still data to read after the current file position.

Eoln (F): Boolean

Returns false if there is still data to read after the current file position, but before the next newline.

SeekEof (F): Boolean

As **Eof**, but first it skips all white spaces.

SeekEoln (F): Boolean

As **Eoln**, but first it skips all white spaces except newline.

Page (F)

Write a form feed to the file. Same as `Write(chr(12))`.

SetTextBuf(F,Buffer)

This associates an I/O buffer of the variable `Buffer` for use with the text file `F`. The default buffer size is 128 bytes but this call will use a buffer of `SizeOf(Buffer)`. `SetTextBuf` is normally called straight after `assign` but you can also use it after `reset` or `rewrite` but before any I/O operations.

SetTextBuf(F,Buffer,Size)

This associates an I/O buffer for use with the text file `F` in the same way as the two parameter version but with a buffer size of `Size` bytes.

Error handling

After all calls to the input/output system an I/O-check routine is called to see if anything went wrong in the last routine. This can be disabled using the directive `{ $I - }` or by disabling the `I/O Checks` item in the `Compiler Settings` box. If IO-checking is disabled, the result of each operation can be checked by using the `IOResult` function. If it returns zero, no error did occur. Note that `IOResult` is a function that only returns its value once, the next time it returns zero. So you should normally call `IOResult` immediately after each i/o operation.

Text File Devices

Whenever a text file is about to be opened, a chain of known 'files' is searched. If you try to open one of these files, you will get access to an internal handler that decides what to do about all your input and output calls. The `Input` and `Output` files use this by default.

You can, of course, overwrite this just by doing a new `reset` and `rewrite` on them.

Chapter 2

Function and Procedure Reference

This chapter is a reference manual for the functions and procedures that are built-in to the compiler or as part of the Crt, Dos and Graph units.

Abs function

Declaration Function Abs(N) : (Same type as parameter)

Function Returns the absolute value of the parameter N.

Remark N is an integer-type or a real-type expression.

See also *Int, Trunc, Round*

Example Program Abs_Demo;

```
Const
  Negative = -1000;
  Positive = 1000;
Begin
  Writeln(Negative, ' becomes ', ABS(Negative));
  Writeln(Positive, ' is still ', ABS(Positive));
End.
```

Addr function

Declaration Function Addr(var Object) : Pointer;

Function Returns the address of the specified object.

Remark The Object parameter can be a variable, procedure or function identifier.

See also SPtr, Ptr

Example Program Addr_Demo;

```
Var
  A : Pointer;
  L : LongInt;

Begin
  A := Addr(L);
  { Another way of doing it is: }
  A := @L;
End.
```

Append procedure

Declaration Procedure Append(var F : text
[;Name:String] [;BufSize:LongInt]);

Function Prepares a text file for appending.

Remark The Append procedure can be used in 3 different ways:

Append(MyFile);

Positions the file pointer at the end of a file that has either already been opened, or has had a file name associated with it using Assign.

Append(MyFile, 'NAME.EXT');

Does the same as ReWrite, but positions the file pointer at the end of the file, rather than deleting it first.


```
Append(MyFile, 'NAME.EXT', 10000);
```

Does the same as the above, but also creates a 10000 byte buffer.

See also *ReWrite, Reset, Assign*

Example Program Append_Data;

```
Var
  DataFile : Text;
  X : Byte;

Begin
  { Create a temporary file. }
  Assign(DataFile,'temp.data');
  Rewrite(DataFile);
  For X := 1 to 100 DO
    Writeln(DataFile,'HELLO WORLD');
  Close(DataFile);

  { Now let's append some data. }
  Append(DataFile);
  Writeln(DataFile,'Last line in file');
  Close(DataFile);
End.
```

Arc procedure Graph

Declaration Procedure Arc(X,Y : Integer; StAngle, EndAngle, Radius : Word);

Function Draws a circular arc.

Remark (X,Y) specifies the centre point, StAngle the start angle, EndAngle the end angle and Radius the radius of the arc.

The arc is drawn using the current aspect ratio settings.

The centre point, the starting point and the ending point of the arc will be returned by a call to GetArcCoords.

See also *SetAspectRatio, Sector, Circle, PieSlice, Rectangle, DrawPoly, Ellipse, Bar, GetArcCoords.*

Example

```
Program Arc_Demo;
Uses Graph, Crt;
Var
  Driver, Mode : Integer;
  C : Char;
Begin
  Driver := DETECT;
  InitGraph(Driver, Mode, '');
  Arc(GetMaxX DIV 2, GetMaxY DIV 2, 30, 330, 45);
  C := ReadKey;
  CloseGraph;
End.
```

ArcTan function

Declaration Function ArcTan(N) : Real;

Function Returns the arctangent of the parameter N.

Remark N is an integer-type or real-type expression. The result is the principal value, in radians, of the argument of N.

See also *ValidReal*

Example Program ArcTan_Demo;

```
Var
  Res : Real;
Begin
  Res := ArcTan(123);
End.
```

Assign procedure

Declaration Assign(var F; name:String);

Function Associates the name of an external file with a file variable.

Remark You should use this procedure before calling `reset`, `rewrite` or `append`. The name string should conform to AmigaDOS conventions.

The assignment of the external name remains in force until another `Assign` statement is used on the same file or the non-standard two parameter version of `reset`, `rewrite` or `append` is used.

See also *Append, Close, Reset, ReWrite*

Example Program Assign_Demo;

Uses DOS;

Var

TextF : Text;

Data : String;

Begin

Assign(TextF, ParamStr(1)); {from the command line}
}

{ \$I- }

Reset(TextF); { Opens the file } { \$I+ }

If (IOresult = 0) then

Begin

While NOT(Eof(TextF)) DO

Begin

Readln(TextF, Data); Writeln(Data);

End;

Close(TextF);

End

Else

Writeln('Can''t open file.');

End.

AssignCrt procedure Crt

Declaration AssignCrt(var F);

Function Associates the a file variable with the Crt unit i/o routines.

Remark This procedure can be used as an alternative to the assign procedure to cause file output to go to the Crt rather than to a file

See also Assign, Close, Reset, ReWrite

Bar procedure Graph

Declaration Procedure Bar(x1, y1, x2, y2 : Integer);

Function Fills a bar in the current fill color and style.

Remark The bar is filled, starting in the upper-left corner specified by (x1,y1), ending in the lower-right corner specified by (x2,y2). The bar is not outlined. To do that, call Rectangle or Bar3D.

See also Bar3D, SetFillStyle, Rectangle, Arc, PieSlice, Sector, DrawPoly, Ellipse

Example

```
Program Bar_Demo;
Uses Graph, Crt;
Var
  Driver, Mode : Integer;
  C : Char;
  Patt : Word;
Begin
  Driver := DETECT;
  InitGraph(Driver, Mode, '');
  For Patt := EmptyFill To CloseDotFill Do
  Begin
    SetFillStyle(Patt, Random(GetMaxColor+1));
    Bar(Patt*10, Patt*10, GetMaxX-(Patt*10), GetMaxY-(Patt*10));
  End;
  C := ReadKey;
  CloseGraph;
End.
```

Bar3D procedure Graph

Declaration Procedure Bar3D(x1,y1,x2,y2 : Integer;
Depth : Word; Top : Boolean);

Function Draws and fills a 3-dimensional bar.

Remark The Depth parameter specifies the depth of the bar,
Top specifies whether or not the top of the bar is to
be drawn.

The bar is drawn in the current line color, and filled
using the current fill pattern and fill color.

See also Bar, Rectangle, Arc, PieSlice, Sector, DrawPoly,
Ellipse, Graph.

Example

```
Program Bar3D_Demo;  
Uses Graph,Crt;  
Var  
  Driver,Mode,X,W : Integer;  
  C : Char;  
Begin  
  Driver := DETECT;  
  InitGraph(Driver,Mode,'');  
  W := GetMaxX DIV 10;  
  For X := 1 to 7 Do  
    Begin  
      SetFillStyle(Random(CloseDotFill+1),Random(GetMaxC  
olor)+1);  
      Bar3D(0+(W*X),Random(GetMaxY),W-  
15+(W*X),GetMaxY,20,TopOn);  
    End;  
    C := ReadKey;  
    CloseGraph;  
  End.
```

BlockRead procedure

Declaration Procedure BlockRead(var F,Buf;
Cnt:Integer [; var Res:Integer]);

Function Reads Cnt bytes from the file F into the variable Buf.

Remark F is a file variable (Typed or Untyped) that has been opened. Cnt specifies the number of bytes to read from the file. The actual number of read bytes will be returned in the optional Res parameter.

If Res is not specified, and Cnt bytes could not be read, an IO-error will occur. Otherwise the number of actually read bytes is returned in Res.

See also BlockWrite, Seek

Example Program BlockRead_Demo;

Uses DOS;

Var

F_In,

F_Out : FILE; { Untyped files }

Buf : Array[1..4096] OF Byte; { Disk buffer }

ActualRead,

ActualWritten : Integer;

Begin

Reset(F_In,ParamStr(1));

{ \$I- } Erase(ParamStr(2)); { \$I+ }

Rewrite(F_Out,ParamStr(2));

Repeat { Copy file In to file Out. }

BlockRead(F_In,Buf,SizeOf(Buf), ActualRead);

BlockWrite(F_Out,Buf,ActualRead, ActualWritten);

Until (ActualRead = 0) OR (ActualWritten <>

ActualRead);

Close(F_In);

Close(F_Out);

End.

BlockWrite procedure

Declaration Procedure BlockWrite(var F, Buf;
Cnt: LongInt [; var Res: Integer]);

Function Writes Cnt bytes from the variable Buf to the file F.

Remark F is a file variable (Typed or Untyped) that has been opened. Cnt specifies the number of bytes to write to the file. The actual number of written bytes will be returned in the optional Res parameter.

Cnt specifies the number of bytes to write to the file. The actual number of written bytes will be returned in the optional Res parameter.

If Res is not specified, and Cnt bytes could not be written, an IO-error will occur. Otherwise the number of actually written bytes is returned in Res.

See also BlockRead, Seek

ChDir procedure DOS

Declaration Procedure ChDir(Dir : DirStr);

Function Type
DirStr = String[67];

Changes the current directory.

Remark This is part of the Dos unit.

See also RmDir, Mkdir, GetDir

Example Program ChDir_Demo;

Uses DOS;

Begin

ChDir(ParamStr(1));

End.

Chr function

Declaration Function Chr(N) : Char;

Function Returns ASCII character number N.

Remark N is an integer type value in the range of 0..255.

See also Ord, Ord4

Example Program Chr_Demo;

```
Var
  B : Byte;
Begin
  For B := 32 to 255 Do
    Write(CHR(b):5);
End.
```

Circle procedure Graph

Declaration Procedure Circle(X,Y : Integer; Radius : Word);

Function Draws a circle.

Remark (X,Y) specifies the centre point and Radius the radius of the circle. The circle is drawn in the current color, using the current aspect ratio settings.

See also SetAspectRatio, Arc, Sector, PieSlice, Rectangle, DrawPoly, Ellipse, Bar

Example Program Circle_Demo;

```
Uses Graph,Crt;
Var
  Driver,Mode,X : Integer;
  C : Char;
Begin
  Driver := DETECT;
  InitGraph(Driver,Mode,'');
  For X := 0 to 100 Do
    Begin
      SetColor(Random(GetMaxColor+1));
```

```

Circle(GetMaxX DIV 2, GetMaxY DIV 2, X);
End;
SetColor(Black);
For X := 100 Downto 0 Do
Circle(GetMaxX DIV 2, GetMaxY DIV 2, X);
C := ReadKey;
CloseGraph;
End.

```

ClearDevice procedure Graph

Declaration Procedure ClearDevice;

Function Clears the entire screen.

Remark The current pointer is also moved to point (0,0).

See also ClearViewport

Example

```

Program ClearDevice_Demo;
Uses Graph,Crt;
Var
  Driver,Mode : Integer;
Begin
  Driver := DETECT;
  InitGraph(Driver,Mode,'');
  Repeat
    Line(0,0,Random(GetMaxX),Random(GetMaxY));
  Until (Random(999) = 1);
  ClearDevice;
  CloseGraph;
End.

```

ClearViewPort procedure Graph

Declaration Procedure ClearViewPort;

Function Clears the current viewport.

Remark Clears the viewport and moves the current pointer to point (0,0). The viewport is filled in the current background color, with the fill pattern set to EmptyFill.

See also ClearDevice, SetViewPort, GetViewSettings

Example

```
Program ClearViewPort_Demo;
Uses Graph,Crt;
Var
  Driver,Mode,
  RanX,RanY : Integer;
  C : Char;
Begin
  Driver := DETECT;
  InitGraph(Driver,Mode,'');
  FloodFill(1,1,1);
  Repeat
    RanX := Random(GetMaxX);
    RanY := Random(GetMaxY);
    SetViewPort(RanX,RanY,RanX+10,RanY+10,ClipOn);
    ClearViewPort;
    Until KeyPressed;
  C := ReadKey;
  CloseGraph;
End.
```

Close procedure

Declaration Procedure Close(var F);

Function Closes an open file.

Remark F is a file variable of any type. Close flushes the file's buffer and releases its handle.

See also Reset, ReWrite, Append, Assign

Example

Program Close_Demo;

Var

Data : Text;

Begin

Assign(Data,'data.tmp');

Rewrite(Data);

Writeln(Data,'About to close this file.');

Close(Data);

End.

CloseGraph procedure Graph

Declaration Procedure CloseGraph;

Function De-initializes and shuts down the Graph unit.

Remark CloseGraph should always be called at the end of your programs if a successful call has previously been made to InitGraph.

CloseGraph does the following:

Closes the screen that has been opened for the Graph unit to use.

Frees the memory that has been allocated by the graph unit.

Closes the Amiga libraries that InitGraph has opened.

See also InitGraph

Example Program CloseGraph_Demo;

Uses Graph;

Var

Driver,Mode : Integer;

Begin

Driver := DETECT;

InitGraph(Driver,Mode,'');

{ Make use of the Graph unit routines. }

CloseGraph;

End.

ClrEol procedure CRT

Declaration Procedure ClrEol;

Function Deletes all characters on the current line, starting at the current cursor position.

Remark The position of the cursor is not changed by the procedure.

See also ClrScr, ClrEos

Example

```
Program ClrEol_Demo;  
Uses Crt;  
Var  
  Y : Byte;  
Begin  
  ClrScr;  
  For Y := 1 to 24 DO  
    Writeln('0123456789');  
  For Y := 1 to 25 DO  
    Begin  
      GoToXY(5,Y); ClrEol;  
    End;  
End.
```

Close procedure

Declaration Procedure Close; (Var F : File);

Function Closes an open file.

Remark I have used the graph unit routines to close the file to illustrate the use of the Close procedure.

See also Read, Rewrite, Append, Assign

ClrEos procedure CRT

Declaration Procedure ClrEos;

Function Clears the CLI window starting at the current cursor location.

Remark The position of the cursor is not changed by the procedure.

See also ClrScr, ClrEol

Example

```
Program ClrEos_Demo;
Uses Crt;
Var
  X : Integer;
Begin
  For X := 1 to 1999 DO
    Begin
      GoToXY(Random(Lo(WindMax))+1,
              Random(Hi(WindMax))+1);
      Write('');
    End;
  GoToXY(1,10);
  ClrEos;
End.
```

ClrScr procedure CRT

Declaration Procedure ClrScr;

Function Clears the screen.

Remark After the call to ClrScr the cursor is placed in the upperleft corner of the window (co-ordinates 1,1).

See also ClrEol, ClrEos

Example

```
Program ClrScr_Demo;
Uses Crt;
Begin
  ClrScr;
End.
```

Concat function

Declaration Function Concat(S1 [, S2, S3 ...]) : String;

Function Concatenates 1 or more strings into 1 string.

Remark Using standard string addition, as for example:

S1 := S1 + S2;

will give the same result as:

S1 := Concat(S1,S2);

See also Copy,Insert,Delete

Example Program Concat_Demo;

```
Var
  A1,A2,Res : String;
Begin
  A1 := 'HELLO '; A2 := 'WORLD';
  Res := Concat(A1,A2);
  WriteLn(Res);
End.
```


Copy function

Declaration Function Copy(FromStr : String; Index, Count : Byte) : String;

Function Returns Count characters from FromStr, starting at position Index.

Remark Copy returns an empty string if index is 0 or greater than the length of FromStr. If Index+Count is greater than the length of FromStr, the Copy function stops at the end of FromStr and returns.

See also Concat, Insert, Delete, Length, Pos

Example Program Copy_Demo;

```
Const
  Main = 'EXTRACT THE WORD HELLO FROM THIS STRING';
Begin
  Writeln(Copy(Main,18,5));
End.
```

Cos function

Declaration Function Cos(N) : Real;

Function Returns the cosine of the parameter.

Remark N is an integer-type or real-type expression. N is assumed to represent an angle in radians.

See also Sin, ValidReal

Example Program Cos_Demo;

```
Var
  R : Real;
Begin
  R := COS(100);
End.
```

Dec procedure

Declaration Procedure Dec(N [; Count]);

Function Decrements an ordinal variable by 1 or by Count.

Remark N and the optional parameter Count are both simple types. If Count is not specified N will be decremented by 1. Otherwise N will be decremented by Count.

See also Inc, Succ

Example Program Dec_Demo;

```
Var
  L : LongInt;
Begin
  L := 100000;
  Repeat
    Dec(L); { Decrease L by 1 }
    Dec(L,2); { Decrease L by 2 }
  Until (L < 0);
End.
```

Delay procedure

Declaration Procedure Delay(Ms : LongInt);

Function Suspends program execution for MS milliseconds.

Example Program Delay_Demo;
Begin
 Writeln('Going to sleep for 5 seconds.');

```
  Delay( 5*1000 );
End.
```

Delete procedure

Declaration Procedure Delete(var S : String; Index, Count : Byte);

Function Deletes Count characters from the string S, starting at position Index.

Remark If Index is 0 or greater than the length of S, no characters are deleted from S.

See also Omit, Copy, Insert, Length, Pos

Example Program Delete_Demo;

```
Var
  S : String;
Begin
  S := 'DELETE THE WORD HELLO FROM THIS STRING';
  Writeln('Before calling delete: ',s);
  Delete(S,17,6);
  Writeln('After calling delete : ',S);
End.
```

DelLine procedure **CRT**

Declaration Procedure DelLine;

Function Deletes the current line from the screen.

Remark All lines following the current line will be scrolled up, and an empty line inserted at the bottom of the CLI window.

See also *InsLine, GotoXY*

Example Program DelLine_Demo;
Uses Crt;

```
Var
  X,Y : Integer;
Begin
  For X := 1 to Lo(WindMax) DO
    For Y := 1 to Hi(WindMax) DO
      Begin
        GoToXY(X,Y); Write('*');
      End;
    GoToXY(30,10);
    Writeln('Press ENTER to go on');
    Readln;
    GoToXY(1,1);
    For Y := 1 to Hi(WindMax) DO
      DelLine;
    End.
```

DetectGraph procedure

Graph

Declaration Procedure DetectGraph(var GraphDriver, GraphMode : Integer);

Function Checks the hardware to find a legal graphics driver and mode.

Remark On the Amiga, HighSpeed Pascal always returns Amiga for the driver and AmigaWB16 as the graphics mode.

Note: DetectGraph is called by the InitGraph procedure if InitGraph receives a driver value of DETECT.

See also InitGraph, GraphResult

Example Program DetectGraph_Demo;
Uses Graph,Crt;
Var
Driver,Mode : Integer;
Begin
Write('The default driver for InitGraph is: ');
DetectGraph(Driver,Mode);
Case Driver of
Amiga: Write('Amiga',);
else: Write('UNKNOWN?');
End;
End.

Dispose procedure

Declaration Procedure Dispose(var P : Pointer);

Function Disposes a dynamic variable.

Remark P is a pointer variable of any type that was previously assigned by the New procedure or was assigned a value by an assignment statement.

After a call to Dispose the variable referenced by P is destroyed and its space in the heap is released.

See also New, FreeMem, GetMem

Example Program Dispose_Demo;

```
Type
  PersonType = RECORD
    Name : String[20];
    Age : Byte;
  END;
Var
  PersonData : ^PersonType;
Begin
  New(PersonData); { Allocate space in heap }
  With PersonData^ DO
    Begin
      Write('Enter name: '); ReadLn(Name);
      Write('Enter age : '); ReadLn(Age);
    End;
  Dispose(PersonData); { Dispose the space }
End.
```

DrawPoly procedure Graph

Declaration Procedure DrawPoly(NumPoints : Word; var PolyPoints);

Function Draws a polygon.

Remark PolyPoints contains the polygon co-ordinates. NumPoints specifies the number of co-ordinates contained in PolyPoints.

Each co-ordinate in PolyPoints consists of an X and a Y value, both of type integer.

Note: In order to draw a closed polygon, the last co-ordinate in PolyPoints must be the same as the first one.

See also FillPoly, SetWriteMode, Arc, Circle, PieSlice, Rectangle, Sector, Ellipse, Bar

Example

```
Program DrawPoly_Demo;
Uses Graph,Crt;
Var
  Driver,Mode : Integer;
  C : Char;
  Poly : Array[1..4] Of PointType;
Begin
  Driver := DETECT;
  InitGraph(Driver,Mode,'');
  Poly[1].X := GetMaxX DIV 2; { Top middle point
of triangle }
  Poly[1].Y := GetMaxY DIV 10;
  Poly[2].X := GetMaxX DIV 10; { Bottom left point
}
  Poly[2].Y := GetMaxY-Poly[1].Y;
  Poly[3].X := GetMaxX-Poly[2].X; { Bottom right
point }
  Poly[3].Y := Poly[2].Y;
  Poly[4] := Poly[1]; { Connect to top point }
  DrawPoly(4,Poly);
  C := ReadKey;
  CloseGraph;
End.
```

Ellipse procedure Graph

Declaration Procedure Ellipse(X, Y: Integer; StAngle, EndAngle, XRad, YRad: Word);

Function Draws an elliptical arc.

Remark (X,Y) specifies the centre point, StAngle the start angle, EndAngle the end angle, XRad the x-radius and YRad the y-radius of the arc.

The arc is drawn in the current color.

The centre point, the starting point and the ending point of the arc will be returned by a call to GetArcCoords.

See also FillEllipse, Arc, Circle, PieSlice, Rectangle, Sector, DrawPoly, Bar, GetArcCoords

Example

```
Program Ellipse_Demo;
Uses Graph,Crt;
Var
  Rad,
  Driver,Mode : Integer;
  C : Char;
Begin
  Driver := DETECT;
  InitGraph(Driver,Mode,'');
  Rad := 0;
  Repeat
    Ellipse(GetMaxX DIV 2,GetMaxY DIV 2,0,360,300-
    Rad,Rad);
    Inc(Rad,10);
  Until (Rad = 300);
  C := ReadKey;
  CloseGraph;
End.
```


EnvCount function **DOS**

Declaration Function EnvCount : Integer;

Function Returns the number of environment strings in the ENV: directory.

Remark Use the functions EnvStr and GetEnv to examine the contents of the environment strings.

Note that this only includes the environment variables in the main ENV: directory; it does not include those within further directories.

See also EnvStr, GetEnv

Example Program EnvCount_Demo;

Uses DOS;

Begin

 Writeln('Number of environment strings is:
 ',EnvCount);

End.

EnvStr function **DOS**

Declaration Function EnvStr(Index : Integer) : String;

Function Returns the environment string number Index.

Remark If Index is 0 or greater than EnvCount, EnvStr will return an empty string. Only the first line of the file is returned.

Note that this only includes the environment variables in the main ENV: directory; it does not include those within further directories. These may be accessed via GetEnv function.

See also EnvCount,GetEnv

Example

Program EnvStr_Demo;

Uses DOS;

```
Var
  Count : Integer;
Begin
  Writeln('Program to display the environment :');
  Writeln;
  For Count := 1 to EnvCount DO
    Writeln( EnvStr(Count) );
  Writeln;
  Writeln('Done.');
```

End.

Eof function

Declaration Function Eof [(var F)] : Boolean;

Function

Returns false if there is still data to read after the current file position. F is a file variable of any type, that has been opened. If F is omitted the standard file Input is assumed.

See also

SeekEof, Eoln

Example

Program Eof_Demo;

Uses DOS;

```
Var
  T : Text;
  D : String;
Begin
  {$I-} Reset(T, ParamStr(1) ); {$I+}
  If (IOresult = 0) then
    Begin
      While NOT(Eof(T)) DO
        Begin
          ReadLn(T,D); Writeln(D);
        End;
      Close(T);
    End
  Else
    Writeln('Error opening file.');
```

End.

Eoln function

Declaration Function Eoln [(var F : text)] : Boolean;

Function Returns false if there is still data to read after the current file position, but before the next new line.

Remark F is a file variable of type text, that has been opened. If F is omitted the standard file Input is assumed.

See also SeekEoln, Eof

Example Program Eoln_Demo;

Uses DOS;

Var

F : Text;

C : Char;

Begin

Reset(F, ParamStr(1));

While NOT(Eoln(F)) DO

Begin

Read(F, C);

Write(C);

End;

Close(F);

End.

Erase procedure

Declaration Procedure Erase(F : String);

Function Erases a file from disk.

Remark Do not attempt to erase an open file.

See also *Rename*

Example Program Erase_Demo;

```
Var
  DummyFile : FILE;
Begin
  { Create a file. }
  Rewrite(DummyFile, 'DUMMY.DTA');
  Close(DummyFile);
  Write('Press RETURN to erase the dummy file.');
```

Readln;

```
  Erase('DUMMY.DTA');
```

End.

Exec procedure **DOS**

Declaration Procedure
Exec(path:PathStr;Command:ComStr);

Function Executes a command with the given command line.

Remark This is implemented via the AmigaDOS unit Execute command. The PathStr and ComStr types are declared in the DOS unit as follows:

```
Type
  PathStr = String[127];
  ComStr = String[127];
```

Example Program Exec_Demo;

```
Uses Dos;
Begin
  Exec('type',ParamStr[1]);
  { shows the file given on the command line }
End.
```

Exit procedure

Declaration Procedure Exit;

Function Exits from the current block of code.

Remark If the current block is the main program, the call to Exit will cause the program to terminate. If the current block is a procedure or a function an exit is made, and control is passed to the caller. Do not confuse this with the Exit_ procedure of the AmigaDOS unit.

See also Halt

Example

```
Program Exit_Demo;  
  Procedure WasteTime;  
  Begin  
    Repeat  
      If (Random(1000) = 999) then EXIT;  
    Until False;  
  End;  
Begin  
  WasteTime;  
End.
```

Exp function

Declaration Function Exp(N) : Real;

Function Returns the exponential of N.

Remark N is an integer-type or real-type expression.

See also Ln, ValidReal

Example

```
Program Exp_Demo;  
  
Var  
  R : Real;  
Begin  
  Write('Enter a number: '); Readln(r);  
  Writeln;  
  Writeln('The exponential is: ',EXP(r));  
End.
```

FExpand function DOS

Declaration Function FExpand(Path : PathStr) : PathStr;

Function Expands a file path into a fully qualified file name.

Remark The file must exist and the path name is returned with the casing as it is on the disk. Wild cards should not be used although directory names may be used.

Suppose the current directory is Hard2:Pascal/docs where Hard2: is the volume name of dh2:. Then FExpand might return the following:

```
FExpand('') = Hard2:Pascal/docs
```

```
FExpand('DOS.PAS.DOC') =  
Hard2:Pascal/docs/Dos.pas.doc
```

```
FExpand('/units/dos.uni') =  
Hard2:Pascal/unitd/Dos.unit
```

```
FExpand('DH2:') = 'Hard2:'
```

FExpand can also expand assigned directories.

See also FExpandLock, FSplit, FindFirst

Example Program FExpand_Demo;

Uses DOS;

Var

Path : PathStr;

Begin

Write('Enter path to expand: ');

Readln(Path);

Path := FExpand(Path);

Writeln('Fully expanded: ', Path);

End.

FExpandLock function DOS

Declaration Function FExpand(Lock : LongInt) :
 PathStr;

Function Finds the fully qualified file name of an AmigaDOS lock.

Remark This is an Amiga specific function, which be used to give the full pathname of a lock that has been obtained from AmigaDOS via the Lock or DupLock functions. Normally you should use the more portable FExpand function.

The full pathname can then be split into its component parts via the FSplit procedure if you wish.

See also FExpand, FSplit, FindFirst

Example Program FExpand_Demo;

 Uses DOS,AmigaDOS;

 Var
 Path : PathStr; MyLock:LongInt;
 Begin
 System.Write('Enter path to expand: ');
 Readln(Path);
 MyLock:=Lock(Path,ACCESS_READ);
 Path := FExpandLock(Path);
 Writeln('Fully expanded: ',Path);
 End.

FilePos function

Declaration Function FilePos(var F) : LongInt;

Function Returns the number of the current record in file F.

Remark F is a file variable (Typed or Untyped) that has been opened. After a call to Reset or ReWrite, FilePos will return 0.

See also Seek, FileSize

Example

```
Program FilePos_Demo;
Uses Crt;
Var
  DataFile : FILE Of Byte;
  B : Byte;

Begin
  { Create a temporary file. }
  Assign(DataFile, 'temp.dta');
  Rewrite(DataFile);
  For B := 1 to 100 DO
    Write(Datafile, B);

  Repeat { Seek around in the file. }
    Seek(DataFile, Random(FileSize(DataFile)));
    WriteLn('Current file position is:
', FilePos(DataFile));
  Until KeyPressed;
  Close(DataFile);
  Erase('temp.dta');
End.
```


FileSize function

Declaration Function FileSize(var F) : LongInt;

Function Returns the number of records in file F.

Remark F is a file variable (Typed or Untyped) that has been opened.

See also FilePos, Seek

Example Program FileSize_Demo;

```
Var
  F : File Of Integer;
  X : Integer;

Begin
  Assign(F,'temp.dta');
  Rewrite(F);
  For X := 1 to 10 DO
    Write(F,X);
  Writeln('Number of records in file:
',FileSize(F));
  Close(F);
  Erase('TEMP.DTA');
End.
```

FillChar procedure

Declaration Procedure FillChar(var Object; Count : LongInt; FillVal : Byte);

Function Fills memory starting at the address specified by Object.

Remark Object is any variable. Count specifies how many bytes of memory that are to be filled out. FillVal specifies the value that is to be used for the fill.

If Count is greater than the size of Object, other data or code may be overwritten, causing unpredictable results.

Therefore the `SizeOf` function should be used to specify the number of bytes to fill.

See also *SizeOf, Move*

Example Program `FillChar_Demo`;

```
Var
  Big : Packed Array[1..30000] OF Byte;
  C : Integer;

Begin
  Writeln('Doing it the slow way.');
```

For C := 1 to 30000 DO

```
  Big[C] := 0;
  Writeln('Doing it the fast way.');
```

FillChar(Big,SizeOf(Big),0);

```
End.
```

FillEllipse procedure Graph

Declaration Procedure `FillEllipse(X, Y: Integer; XRadius, YRadius: Word);`

Function Draws and fills an ellipse.

Remark (X,Y) specifies the centre point, `XRadius` the x-radius and `YRadius` the y-radius of the ellipse. The ellipse is drawn using the current color, and filled using the current fill pattern and fill color.

See also *SetFillStyle, SetFillPattern, Ellipse, Arc, Circle, PieSlice, Rectangle, Sector, DrawPoly, Bar*

Example Program `FillEllipse_Demo`;

```
Uses Graph,Crt;
Var
  Driver,Mode : Integer;
  C : Char;

Begin
  Driver := DETECT;
  InitGraph(Driver,Mode,'');
  FillEllipse(GetMaxX DIV 2,GetMaxY DIV 2,50,100);
  C := ReadKey;
  CloseGraph;
End.
```

FillPoly procedure Graph

Declaration Procedure FillPoly(NumPoints : Word; var PolyPoints);

Function Draws and fills a polygon.

Remark PolyPoints contains the polygon co-ordinates. NumPoints specifies the number of co-ordinates contained in PolyPoints.

Each co-ordinate in PolyPoints consists of an X and a Y value, both of type Integer.

The polygon is drawn in the current color, and filled using the current fill settings.

See also DrawPoly, SetFillStyle, Arc, Circle, PieSlice, Rectangle, Sector, Ellipse, Bar, Graph

Example Program FillPoly_Demo;
Uses Graph,Crt;
Var
 Driver,Mode : Integer;
 C : Char;
 Poly : Array[1..3] Of PointType;
Begin
 Driver := DETECT;
 InitGraph(Driver,Mode,'');
 SetFillStyle(XHatchFill,GetMaxColor);
 Poly[1].X := GetMaxX DIV 2; { Top middle point
of triangle }
 Poly[1].Y := GetMaxY DIV 10;
 Poly[2].X := GetMaxX DIV 10; { Bottom left point }
 Poly[2].Y := GetMaxY-Poly[1].Y;
 Poly[3].X := GetMaxX-Poly[2].X; { Bottom right
point }
 Poly[3].Y := Poly[2].Y;
 FillPoly(3,Poly);
 C := ReadKey;
 CloseGraph;
End.

FindFirst procedure **DOS**

Declaration Procedure FindFirst(Path:String;
Attr:Byte; var S:SearchRec);

Function Searches a directory for the first entry matching a specified file spec and set of attributes.

Remark Path is a path to a directory on a specified drive. The path consists of a directory name, which is optional, and a file spec.

Example There are two sorts of wildcard characters that may be used, AmigaDOS ones and MS-DOS ones.

The AmigaDOS wildcards are more general with #,?, \ and ranges enclosed in parentheses () supported. The MS-DOS wildcards are limited to * and ?. ? means any single character can replace the filename and * means a sequence of any number of characters.

Naturally the MS-DOS variant has the advantage of making it ease to move programs to and from Turbo Pascal on the PC and HighSpeed Pascal on the Atari.

By default AmigaDOS conventions are used but you can change this to the MS-DOS wildcards using:

```
PatternProc=@MsDosPattern;
```

and back to the AmigaDOS standard using:

```
PatternProc=@ADosPattern;
```

The Attr parameter specifies the special files to search for (in addition to all normal files).

The file attributes are declared in the DOS unit as follows:

```
DeleteFlag = $0001;      {Delete allowed}
ExecuteFlag = $0002;      {Run allowed}
WriteFlag   = $0004;      {Write allowed}
ReadFlag    = $0008;      {Read allowed}
Archive     = $0010;      {Backup flag}
```

```

PureFlag   = $0020;      {Residentable programs}
ScriptFlag = $0040;      {Set for Script files}
InfoFlag   = $0100;      {Set for .info files}
VolumeID   = $0200;      {First file in each dir}
Directory  = $0400;      {A directory}
AnyFile    = $0FFF;      {All flags together}

```

Upon return `FindFirst` places the result of the search in the `S` variable. `S` is of the type `SearchRec`, which is declared in the DOS unit:

```

SearchRec = RECORD
  Lock   : LongInt;      {*Internal use only}
  Attr   : Integer;      {Flag of file found}
  Size   : LongInt;      {Size of found file}
  Time   : PackedTime;   {Date&Time}
  Name   : NameStr;      File found}
  Reserved : tFileInfoBlock; {* }
  ResAttr  : Integer;     {* Attr wanted}
  ResPattern : NameStr;   {* Pattern wanted}
END;

```

After a call to `FindFirst` the variable `DosError` must be checked. If it is not 0, the contents of the `S` parameter will be garbage.

After finding the first file using this procedure, you can find subsequent files using `FindNext`. If you do not read the entire directory until no more files are found you *must* call `StopFindFirst` when you have finished reading the directory.

`StopFindFirst` is not used under MS-DOS or on the Atari but is required on the Amiga to ensure that the AmigaDOS Lock that is used to read the directory is freed. If you read a directory to its end using `FindNext`, as in 95% of the uses of these procedures, you do not need to call `StopFindFirst` as the Lock will be freed automatically.

See also *FindNext, StopFindFirst, UnPackTime*

Example Program `FindFirst_FindNext_Demo`;

Uses DOS;

Var

DosData : SearchRec;

```

Begin
  FindFirst( '#?' , AnyFile, DosData );
  While (DosError = 0) DO
    Begin
      With DosData DO
        WriteLn(Name);
        FindNext(DosData);
    End;
  End.

```

FindNext procedure DOS

Declaration Procedure FindNext(var S : SearchRec)

Function Finds the next entry matching the specifications given in a previous call to FindFirst.

Remark The S parameter must be the same one that was used in the call to FindFirst.

After a call to FindNext the variable DosError must be checked. If it is not 0, the contents of the S parameter will be garbage as the end of the directory will have been reached.

If you do not read the entire directory until no more files are found you *must* call StopFindFirst when you have finished reading the directory.

StopFindFirst is not used under MS-DOS or on the Atari but is required on the Amiga to ensure that the AmigaDOS Lock that is used to read the directory is freed. If you read a directory to its end using FindNext, as in 95% of the uses of these procedures, you do not need to call StopFindFirst as the Lock will be freed automatically.

See also FindFirst, StopFindFirst, UnPackTime

FloodFill procedure Graph

Declaration Procedure FloodFill(X, Y : Integer;
Border : Word);

Function Fills an area surrounded by a specified color.

Remark (X,Y) specifies the point that is filled first. Border specifies the color that surrounds the area to fill.

The area is filled in the current fill color, using the current fill pattern.

See also FillEllipse, FillPoly, Bar, Bar3D, PieSlice, SetFillPattern, SetFillStyle, GetFillPattern, GetFillSettings

Example

```
Program FloodFill_Demo;  
Uses Graph,Crt;  
Var  
  Driver,Mode : Integer;  
  C : Char;  
Begin  
  Driver := DETECT;  
  InitGraph(Driver,Mode,'');  
  SetColor(Blue);  
  Rectangle(10,10,100,100); { Make BLUE  
rectangle }  
  SetFillStyle(SolidFill,Green);  
  FloodFill(11,11,Blue); { Flood until BLUE  
is found }  
  C := ReadKey;  
  CloseGraph;  
End.
```


FreeImage procedure Graph

Declaration Procedure FreeImage(var BitMap; size : longint);

Function Frees memory allocated via a calls to GetMem and GetImage.

Remark BitMap should correspond to the same parameter of GetImage and size should correspond to the appropriate value returned by ImageSize.

This procedure *must* be called instead of FreeMem when you want to free an image buffer that has been used with PutImage/GetImage.

Note that this call is unique to the Amiga version. It is required because this implementation uses part of the memory for an internal structure which must be properly freed.

See also GetImage, PutImage, ImageSize

FreeMem procedure

Declaration Procedure FreeMem(var P : Pointer; Size : LongInt);

Function Disposes a dynamic variable of a given size.

Remark P is a pointer variable of any type that was previously assigned by the GetMem procedure or was assigned a value by an assignment statement. Size specifies the size of the dynamic variable to dispose.

Size must be equal to the size specified in the call to GetMem.

Do not confuse this procedure with the Exec call Freemem_. Note the terminating underscore.

See also GetMem, New, Dispose

Example

```
Program FreeMem_Demo;

Var
  Data : Pointer;
  F : FILE;
Begin
  GetMem(Data,1024); { Allocate 1K in the heap }
  Reset(F3ParamStr(1));
  BlockRead(F,Data^,SizeOf(Data^));
  Close(F);
  FreeMem(Data,1024);
End.
```

FSplit procedure DOS

Declaration Procedure FSplit(P:PathStr;var
D:DirStr;var N:NameStr;var E:ExtStr);

Function Splits a file path up into its 3 components:

- Remark**
- The directory specification.
 - The file name.
 - The file extension.

Any of the 3 components can be empty on return, if the file path did not contain that component on entry.

The PathStr, DirStr, NameStr and ExtStr types are declared in the DOS unit as follows:

```
Type
  PathStr = String[127];
  DirStr = String[127];
  NameStr = String[31];
  ExtStr = String[15];
```

See also FExpand, GetDir

Example Program FSplit_Demo;

Uses DOS;

```
Var
  FullPath : PathStr;
```

```

Dir      : DirStr;
Name     : NameStr;
Ext      : ExtStr;
Begin
  Repeat
    Write('Enter path to split: ');
    Readln(FullPath);
    FSplit(FullPath,Dir,Name,Ext);
    Writeln;
    Writeln(FullPath,' consists of:');
    Writeln;
    Writeln('The directory: ',Dir);
    Writeln('The file name : ',Name);
    Writeln('With the extension: ',Ext);
  Until (FullPath = '');
End.

```

GetArcCoords procedure Graph

Declaration Procedure GetArcCoords(var ArcCoords : ArcCoordsType);

Function Returns information about the last call to Arc or Ellipse.

Remark The ArcCoordsType record is declared in the Graph unit as:

```

Type
ArcCoordsType = Record
    X,Y      : Integer;
    XStart   : Integer;
    YStart   : Integer;
    XEnd     : Integer;
    YEnd     : Integer;
End;

```

(X,Y) is the centre point, (XStart,YStart) is the starting point and (XEnd,YEnd) is the end point of the last arc or ellipse drawn.

See also Arc, Ellipse, Sector, PieSlice

Example

```
Program GetArcCoords_Demo;  
Uses Graph,Crt;  
Var  
  Driver,Mode : Integer;  
  C : Char;  
  ArcInfo : ArcCoordsType;  
Begin  
  Driver := DETECT;  
  InitGraph(Driver,Mode,'');  
  Arc(GetMaxX DIV 2,GetMaxY DIV 2,30,330,45);  
  GetArcCoords(ArcInfo);  
  With ArcInfo Do  
  Begin  
    Line(X,Y,XStart,YStart);  
    Line(X,Y,XEnd,YEnd);  
  End;  
  C := ReadKey;  
  CloseGraph;  
End.
```

GetAspectRatio procedure Graph

Declaration Procedure GetAspectRatio(var Xasp, Yasp : Word);

Function Returns the X and Y value used to calculate the aspect ratio.

Remark The aspect ratio is used by Arc, Circle and PieSlice to ensure that the circular figures will appear on the screen as being round and not egg-shaped.

The aspect ratios depend on the screen mode being used. The default for Yasp is 10000; for a standard PAL non-interlaced screen the default for Xasp is 18750.

See also SetAspectRatio, Arc, Circle, PieSlice

Example Program GetAspectRatio_Demo;
Uses Graph,Crt;
Var
 Driver,Mode : Integer;
 C : Char;

```

Xasp,Yasp : Word;
Begin
  Driver := DETECT;
  InitGraph(Driver,Mode,'');
  SetTextJustify(CenterText,CenterText);
  OutTextXY(100,100,'Circle');
  Circle(100,100,45);
  GetAspectRatio(Xasp,Yasp);
  Dec(Yasp,5000);
  SetAspectRatio(Xasp,Yasp);
  OutTextXY(200,100,'Egg');
  Circle(200,100,45);
  C := ReadKey;
  CloseGraph;
End.

```

GetBkColor function Graph

Declaration Function GetBkColor : Word;

Function Returns the number of the current background color.

Remark The range of the returned value depends on the current graphics mode.

See also *GetBkColor, GetColor*

Example Program GetBkColor_Demo;

Uses Graph,Crt,UtilUnit;

Var

Driver,Mode : Integer;

C : Char;

Begin

Driver := DETECT;

InitGraph(Driver,Mode,'');

OutText('Current background color is
' + Int2Str(GetBkColor));

C := ReadKey;

CloseGraph;

End.

GetColor function Graph

Declaration Function GetColor : Word;

Function Returns the current foreground color.

See also SetColor, GetBkColor

Example

```
Program GetColor_Demo;
Uses Graph,Crt,UtilUnit;
Var
  Driver,Mode : Integer;
  C : Char;
Begin
  Driver := DETECT;
  InitGraph(Driver,Mode,'');
  OutText('Current color is: '+Int2Str(GetColor));
  C := ReadKey;
  CloseGraph;
End.
```

GetDate procedure DOS

Declaration Procedure GetDate(var Year, Month, Day, DayOfWeek : Word);

Function Returns the current date according to the operating system.

Remark The ranges of the returned values are:

Year	1980..2099
Month	1..12
Day	1..31
DayOfWeek	0..6 (Where 0 = Sunday, 1 = Monday)

See also SetDate,GetTime

Example Program Show_Day_Of_Week;

Uses DOS;

Var

Year,Month,Day,DayOfWeek : Word;

Begin

```

GetDate(Year,Month,Day,DayOfWeek);
Write('Today is ');
CASE DayOfWeek OF
  0 : Write('Sunday');
  1 : Write('Monday');
  2 : Write('Tuesday');
  3 : Write('Wednesday');
  4 : Write('Thursday');
  5 : Write('Friday');
  6 : Write('Saturday');
END;
Writeln('.');
End.

```

GetDefaultPalette procedure Graph

Declaration Procedure GetDefaultPalette(var Palette : PaletteType);

Function Returns the Graph unit default palette.

Remark The PaletteType record is declared in the Graph unit as:

```

PaletteType = Record
  Size: Integer;
  Colors : Array[0..MaxColors] Of
    ShortInt;
end;

```

Upon return from GetDefaultPalette the Palette parameter will contain the palette color numbers, as they were when the Graph unit was initialized.

See also *GetPalette, SetAllPalette, SetPalette, SetRGBPalette*

Example Program GetDefaultPalette_Demo;
Uses Graph,Crt;
Var
 Driver,Mode,X : Integer;
 C : Char;
 PalData : PaletteType;
Begin

```

Driver := DETECT;
InitGraph(Driver,Mode,'');
{ Draw lines in random colors }
For X := 1 to 200 Do
Begin
SetColor(Random(GetMaxColor+1));
Line(0,0,Random(GetMaxX),Random(GetMaxY));
End;
{ Change the palette colors }
Repeat
SetPalette(Random(GetMaxColor)+1,Random(GetMaxColor)+1);
Until KeyPressed;
C := ReadKey;
GetDefaultPalette(PalData); { Get original
palette }
SetAllPalette(PalData); { Restore palette }
C := ReadKey;
CloseGraph;
End.

```

GetDir procedure DOS

Declaration Procedure GetDir(var Dir : DirStr);

Function Returns the current directory.

Remark Note that under MS-DOS or on the Atari, this procedure has an additional drive parameter. This is omitted on the Amiga as different devices don't have their own current directory. Similarly the GetDrive function of Turbo Pascal is not needed as the device name is returned at the start of the string returned by GetDir.

See also ChDir,RmDir,MkDir,FSplit

Example Program GetDir_Demo;

Uses DOS;

```

Var
CurDir : DirStr;
Begin
GetDir(CurDir);
Writeln('Current dir is: ',CurDir);
End.

```

GetDriverName procedure Graph

Declaration Function GetDriverName : string;

Function Returns the name of the active graphics driver.

Remark When using the Amiga driver the returned string will be Amiga.

See also GetModeName, GetMaxMode, GetGraphMode, GetModeRange, SetGraphMode

Example

```
Program GetDriverName_Demo;  
Uses Graph,Crt;  
Var  
  Driver,Mode : Integer;  
  C : Char;  
Begin  
  Driver := DETECT;  
  InitGraph(Driver,Mode,'');  
  SetTextJustify(CenterText,CenterText);  
  OutTextXY(GetMaxX DIV 2,GetMaxY DIV  
2,GetDriverName);  
  C := ReadKey;  
  CloseGraph;  
End.
```

GetEnv function DOS

Declaration Function GetEnv(MatchStr:String) :String;

Function Returns the value of a specific environment string.

Remark If no match is found for MatchStr,GetEnv will return an empty string.

See also EnvCount,EnvStr

Example

```
Program GetEnv_Demo;  
Uses DOS;  
Begin  
  Writeln('Current PATH is: ', GetEnv('PATH'));  
End.
```


GetFAttr procedure DOS

Declaration Procedure GetFAttr(Fil : PathStr; var
Attr : Integer);

Function Gets the attributes of a file.

Remark The file attributes and PathStr are declared in the
DOS unit:

Const

```
DeleteFlag = $0001;        {Delete allowed}
ExecuteFlag        = $0002;        {Run allowed}
WriteFlag        = $0004;        {Write allowed}
ReadFlag        = $0008;        {Read allowed}
Archive        = $0010;        {Backup flag}
PureFlag        = $0020;        {Residentable programs}
ScriptFlag        = $0040;        {Set for Script files}
InfoFlag        = $0100;        {Set for .info files}
VolumeID        = $0200;        {First file in each dir}
Directory        = $0400;        {A directory}
AnyFile        = $0FFF;        {All flags together}
```

Type

PathStr = String[127];

Note that the attributes are Amiga specific but have some relevance in the MS-DOS world. See the description of the DOS unit constants for details.

See also SetFAttr

Example Program GetFAttr_Demo;

Uses DOS;

Var

Attrs : Integer;

Begin

```
If (ParamCount = 0) then HALT;
GetFAttr( ParamStr(1), Attrs );
If (DosError = 0) then
Begin
  If (Attrs AND DeleteFlag > 0) then
    WriteLn('Delete Allowed');
```

```

If (Attrs AND ExecuteFlag > 0) then
  WriteLn('Executable');
If (Attrs AND WriteFlag > 0) then
  WriteLn('Write Allowed');
If (Attrs AND PureFlag > 0) then
  WriteLn('Pure');
If (Attrs AND ScriptFlag > 0) then
  WriteLn('Script');
If (Attrs AND VolumeID > 0) then
  WriteLn('Info file');
If (Attrs AND Directory > 0) then
  WriteLn('Directory');
If (Attrs AND Archive > 0) then
  WriteLn('Archive');
End
Else
  WriteLn('No file name specified.');
```

End.

Note that the attributes are Amiga specific but have some relevance in the MS-DOS world. See the discussion of the `FileAttributes` function.

Declaration	<code>Function GetEnvStr: String; (String)</code>
Function	Returns the value of a specified environment string.
Remark	If no match is found for <code>MatchStr</code> , <code>GetEnv</code> will return an empty string.
See also	<code>EnvCount</code> , <code>EnvStr</code>
Example	<pre> If (ParamCount = 0) then HALT; GetEnv('ParamCount'); If (ParamCount = 0) then GetEnv('ParamCount'); WriteLn('ParamCount');</pre>

GetFillPattern procedure

Graph

Declaration Procedure GetFillPattern(var FillPattern : FillPatternType);

Function Returns the current user-defined fill pattern.

Remark The FillPatternType array is declared in the Graph unit as:

Type
FillPatternType = Packed Array[1..8] Of Byte;

See also *GetFillSettings, GetFillPattern, SetFillStyle, FloodFill, FillEllipse, FillPoly, Bar, Bar3D, PieSlice*

Example

```
Program GetFillPattern_Demo;
Uses Graph,Crt;
Var
  Driver,Mode,X : Integer;
  C : Char;
  PatternData : FillPatternType;
Procedure WriteBinaryByte( Byt : Byte );
Var Bit : Byte;
Begin
  For Bit := 0 to 7 Do
  Begin
    If (Byt AND 1 > 0) then
      OutText('1')
    Else
      OutText('0');
    Byt := Byt SHR 1;
  End;
End;
Begin
  Driver := DETECT;
  InitGraph(Driver,Mode,'');
  GetFillPattern(PatternData);
  OutText('User-defined fill pattern is: ');
  For X := 1 to 8 Do
  Begin
    MoveTo(0,(X+2)*TextHeight('1'));
    WriteBinaryByte(PatternData[X]);
  End;
  C := ReadKey;
  CloseGraph;
End.
```

GetFillSettings procedure Graph

Declaration Procedure GetFillSettings(var FillInfo : FillSettingsType);

Function Returns the current fill settings.

Remark The FillSettingsType record is declared in the Graph unit as:

Type
FillSettingsType= Record
Pattern : Word;
Color : Word;
End;

Upon return from GetFillSettings, FillInfo will contain the currently active fill pattern and fill color.

See also *GetFillPattern, SetFillStyle, SetFillPattern, FloodFill, FillEllipse, FillPoly, Bar, Bar3D, PieSlice*

Example

```
Program GetFillSettings_Demo;
Uses Graph, Crt, UtilUnit;
Var
  Driver, Mode : Integer;
  C : Char;
  FillInfo : FillSettingsType;
Begin
  Driver := DETECT;
  InitGraph(Driver, Mode, '');
  GetFillSettings(FillInfo);
  With FillInfo Do
  Begin
    OutTextXY(0, 0, 'Current fill pattern is:
'+Int2Str(Pattern));
    OutTextXY(0, TextHeight('M'),
'Current fill color is : '+Int2Str(Color));
  End;
  C := ReadKey;
  CloseGraph;
End.
```

GetFTime procedure DOS

Declaration Procedure GetFTime(var F; var Time : PackedTime);

Function Gets the time and date of a file.

Remark The F parameter is a file variable (Typed, Untyped or Text) that has been either Reset, Rewritten or Appended.

The returned PackedTime variable can be unpacked, using the UnPackTime procedure.

Note that under MS-DOS and on the Atari the PackedTime parameter is of type LongInt. Normally however you need only change the type of the variable you are using as the PackTime and UnpackTime procedures work in the same way.

See also SetFTime, UnPackTime, Reset, ReWrite, Append

Example

```
Program GetFTime_Demo;
Uses DOS;
Var
  DiskFile : TEXT;
  Name : String;
  DatTim : DateTime;
  PackData : PackedTime;
Begin
  Write('Enter name of file to work with: ');
  Readln(Name);
  {$I-} Reset(DiskFile, Name); {$I+}
  If (IOresult = 0) then
    Begin
      GetFTime( DiskFile, PackData );
      UnpackTime(PackData, DatTim);
      With DatTim DO
        Begin
          Writeln('Date of update/creation is: ', Day, '-',
            'Month, ', '/', 'Year);
          Writeln('Time of update/creation is:
            ', Hour, '.', 'Min, ', ' ', 'Sec);
        End;
      Close(DiskFile);
      Writeln('Done.');
```

```
    End
  Else
    Writeln('Error while opening file.');
```

```
End.
```

GetGraphMode procedure

Graph

Declaration Function GetGraphMode : Integer;

Function Returns the current graphics mode.

Remark When using the Amiga driver, the returned result will be one of CGAC0, CGAC1, CGAC2, CGAC3, CGAHi, MCGAHi, EGALo, EGAHi, EGA64Hi, EGAMonoHi, VGAHi, PC3270Hi, StLow, StMedium, StHigh, AmigaWB, AmigaWB16, AmigaLo32, AmigaCustom.

See also *GetMaxMode, GetDriverName, GetModeName, GetModeRange, SetGraphMode*

Example

```
Program GetGraphMode_Demo;
Uses Graph, Crt;
Var
  Driver, Mode : Integer;
  C : Char;
Begin
  Driver := DETECT;
  InitGraph(Driver, Mode, '');
  OutText('Current mode is: ');
  Case GetGraphMode Of
    CGAC0: outtext('CGAC0');
    CGAC1: outtext('CGAC1');
    CGAC2: outtext('CGAC2');
    CGAC3: outtext('CGAC3');
    CGAHi: outtext('CGAHi');
    MCGAHi: outtext('MCGAHi');
    EGALo: outtext('EGALo');
    EGAHi: outtext('EGAHi');
    EGA64Hi: outtext('EGA64Hi');
    EGAMonoHi: outtext('EGAMonoHi');
    VGAHi: outtext('VGAHi');
    PC3270Hi: outtext('PC3270Hi');
    StLow: outtext('StLow');
    StMedium: outtext('StMedium');
    StHigh: outtext('StHigh');
    AmigaWB: outtext('AmigaWB');
    AmigaWB16: outtext('AmigaWB16');
    AmigaLo32: outtext('AmigaLo32');
    AmigaCustom: outtext('AmigaCustom');
  End;
  C := ReadKey;
  CloseGraph;
End.
```

GetImage procedure Graph

Declaration Procedure GetImage(x1, y1, x2, y2 : Integer; var BitMap);

Function Stores a specified screen area in a buffer.

Remark (x1,y1), (x2,y2) define the rectangular area to store. BitMap is the buffer to use for the storage.

If BitMap is too small to hold the defined area, other data or code will be overwritten.

Thus you should *always* use the ImageSize function to determine the required size of BitMap and allocate it using GetMem. This should then be freed via FreeImage.

See also ImageSize, PutImage, FreeImage, ImageSize

Example

```
Program Image_Demo;
Uses Graph;
Const
  DelayTime = 10;
Var
  Driver, Mode : Integer;
  C : Char;
  Storage : Pointer;
  Size : LongInt;
Procedure DrawImage;
Var
  ArcInfo : ArcCoordsType;
Begin
  Arc(30,30,30,330,30);
  GetArcCoords(ArcInfo);
  With ArcInfo Do
    Begin
      Line(X,Y,XStart,YStart);
      Line(X,Y,XEnd,YEnd);
    End;
End; { DrawImage }
Procedure MoveImage;
Var
  X : Integer;
Begin
  ClearDevice;
  X := 60;
```

```

Repeat
PutImage(X,GetMaxY DIV 2,Storage^, CopyPut);
Delay(DelayTime);
PutImage(X,GetMaxY DIV 2,Storage^, XorPut);
If (X >= GetMaxX - 60) then
X := 60;
Inc(X,10);
Until KeyPressed;
End; { MoveImage }
Begin
Driver := DETECT;
InitGraph(Driver,Mode,'');
DrawImage;
Size := ImageSize(0,0,60,60);
If (MaxAvail > Size) then
GetMem(Storage,Size)
Else
Halt;
GetImage(0,0,60,60,Storage^);
MoveImage;
FreeImage(Storage,Size);
CloseGraph;
End.

```

GetLineSettings procedure Graph

Declaration Procedure GetLineSettings(var LineInfo : LineSettingsType);

Function Returns the current style, pattern and width of lines.

Remark The LineSettingsType record is declared in the Graph unit as:

```

Type
LineSettingsType= Record
LineStyle : Word;
Pattern : Word;
Thickness : Word;
End;

```


LineStyle contains the number of the current line style. It will always be in the range **SolidLn..UserBitLn**. **Pattern** specifies the user-defined line pattern.

Thickness specifies the thickness of lines. It will always be either **NormWidth** or **ThickWidth**. Sadly this is not supported by the Amiga's operating system and so all lines appear as the same width.

See also

SetLineStyle, SetWriteMode, Line, LineTo, LineRel

Example

```
Program GetLineSettings_Demo;
Uses Graph,Crt,UtilUnit;
Var
  Driver,Mode : Integer;
  C : Char;
  LineInfo : LineSettingsType;
  Th : Integer;
Begin
  Driver := DETECT;
  InitGraph(Driver,Mode,'');
  Th := TextHeight('M');
  GetLineSettings(LineInfo);
  With LineInfo Do
  Begin
    OutTextXY(0,Th*1,'Current style is      :
'+Int2Str(LineStyle));
    OutTextXY(0,Th*2,'Current pattern is    :
'+Int2Str(Pattern));
    OutTextXY(0,Th*3,'Current thickness is:
'+Int2Str(ThickNess));
  End;
  C := ReadKey;
  CloseGraph;
End.
```

GetMaxColor function Graph

Declaration Function GetMaxColor : Word;

Function Returns the highest legal color number.

Remark The range of the value returned depends on the number of planes supported by the given mode, as shown in the table below:

Name	Planes	GetMaxColor
CGAHi, MCGAHi, EGAMonoHi, PC3270Hi, StHigh	1	1
CGAC0, CGAC1, CGAC2, CGAC3, EGA64Hi, StMedium, AmigaWB	2	3
EGAHi	3	7
EGALo, VGAHi, StLow, AmigaWB16	4	15
AmigaLo32	5	31

When using **AmigaCustom** mode, the number of planes and available colours varies, as set via the **SetCustomMode** procedure.

See also *SetColor, SetBkColor*

Example

```
Program GetMaxColor_Demo;  
Uses Graph,Crt,UtilUnit;  
Var  
  Driver,Mode : Integer;  
  C : Char;  
Begin  
  Driver := DETECT;  
  InitGraph(Driver,Mode,'');  
  OutText('Maximum color is:  
' + Int2Str(GetMaxColor));  
  C := ReadKey;  
  CloseGraph;  
End.
```

GetMaxMode function

Graph

Declaration Function GetMaxMode : Integer;

Function Returns the highest mode number for the currently active driver.

Remark GetMaxMode can be used together with SetGraphMode, to determine the highest possible graphics mode for the currently active driver.

The result returned by GetMaxMode with the Amiga driver is AmigaCustom.

See also GetGraphMode, GetDriverName, GetModeName, GetModeRange, SetGraphMode

Example

```
Program GetMaxMode_Demo;
Uses Graph,Crt,UtilUnit;
Var
  Driver,Mode : Integer;
  C : Char;
Begin
  Driver := DETECT;
  InitGraph(Driver,Mode,'');
  OutText('Maximum mode for '+GetDriverName+' is
'+Int2Str(GetMaxMode));
  C := ReadKey;
  CloseGraph;
End.
```

GetMaxX function Graph

Declaration Function GetMaxX : Integer;

Function Returns the highest possible X co-ordinate.

Remark The result returned by GetMaxY depends on the current graphics driver and mode:

Name	Result
CGAC0	199
CGAC1	199
CGAC2	199
CGAC3	199
CGAHi	199
MCGAHi	479
EGALo	199
EGAHi	349
EGA64Hi	349
EGAMonoHi	349
VGAHi	479
PC3270Hi	256
StLow	199
StMedium	399
StHigh	399
AmigaWB	255
AmigaWB16	255
AmigaLo32	255

The result will be different from the above listed for the Amiga modes on NTSC machines or if over-scan is in use. The GetMaxX value for an AmigaCustom mode screen depends on how it has been set up using SetCustomMode.

See also GetMaxY, GetX, GetY

Example Program GetMaxX_Demo;
 Uses Graph,Crt;

```

Var
  Driver,Mode : Integer;
  C : Char;
Begin
  Driver := DETECT;
  InitGraph(Driver,Mode,'');
  Line(0,0,GetMaxX,0);
  C := ReadKey;
  CloseGraph;
End.

```

GetMaxY function Graph

Declaration Function GetMaxY : Integer;

Function Returns the highest possible Y co-ordinate.

Remark The result returned by GetMaxX depends on the current graphics driver and mode:

Name	Result
CGACO	319
CGAC1	319
CGAC2	319
CGAC3	319
CGAHi	639
MCGAHi	639
EGALo	639
EGAHi	639
EGA64Hi	639
EGAMonoHi	639
VGAHi	639
PC3270Hi	639
StLow	319
StMedium	639
StHigh	639
AmigaWB	639
AmigaWB16	639
AmigaLo32	319

The result can be different from the above listed for the Amiga modes if over-scan is in use. The `GetMaxY` value for an `AmigaCustom` mode screen depends on how it has been set up using `SetCustomMode`.

See also `GetMaxX`, `GetX`, `GetY`

Example

```
Program GetMaxY_Demo;
Uses Graph,Crt;
Var
  Driver,Mode : Integer;
  : Char;
Begin
  Driver := DETECT;
  InitGraph(Driver,Mode,'');
  Line(0,0,0,GetMaxY);
  C := ReadKey;
  CloseGraph;
End.
```

GetMem procedure

Declaration Procedure `GetMem`(var `P` : `Pointer`; `Size` : `LongInt`);

Function Creates a new dynamic variable and sets `P` to point to it.

Remark `P` is a pointer variable of any type. `Size` specifies the size in bytes of the dynamic variable to allocate.

See also `FreeMem`, `Dispose`, `New`

Example

```
Program FreeMem_Demo;

Var
  Data : Pointer;
  F : FILE;
Begin
  GetMem(Data,1024); { Allocate 1K in the heap }
  Reset(F3ParamStr(1));
  BlockRead(F,Data^,SizeOf(Data^));
  Close(F);
  FreeMem(Data,1024);
End.
```

GetModeName function

Graph

Declaration Function GetModeName(ModeNumber : Integer) : string;

Function Returns the name of the active graphics mode.

Remark GetModeName will return a string consisting of:

- The driver name
- The horizontal resolution
- The vertical resolution
- The number of colors

Thus using the default AmigaWB16 mode on a standard PAL machine we will have:

Amiga 640x256, 16 color

See also *GetDriverName, GetMaxMode, GetGraphMode, GetModeRange, SetGraphMode*

Example

```
Program GetModeName_Demo;
Uses Graph,Crt;
Var
  Driver,Mode : Integer;
  C : Char;
Begin
  Driver := DETECT;
  InitGraph(Driver,Mode,'');
  OutText('Current mode is: '+GetModeName(Mode));
  C := ReadKey;
  CloseGraph;
End.
```

GetModeRange procedure Graph

Declaration Procedure
GetModeRange (GraphDriver: Integer; var
LoMode, HiMode: Integer);

Function Returns the mode limits for a graphics driver.

Remark GetModeRange returns the minimum and the maximum graphics mode for a specified graphics driver.

If GraphDriver contains an unknown driver, LoMode and HiMode will both be set to the value -1 (GrNoInitGraph).

Otherwise LoMode will be set to CGAC0 and HiMode to AmigaCustom.

See also GetMaxMode, GetGraphMode, GetDriverName, GetModeName, SetGraphMode

Example

```
Program GetModeRange_Demo;  
Uses Graph, Crt;  
Var  
  Min, Max,  
  Driver, Mode : Integer;  
  C : Char;  
Begin  
  DetectGraph(Driver, Mode);  
  WriteLn('Driver is: ', Driver);  
  GetModeRange(Driver, Min, Max);  
  WriteLn('Range is: ', Min, '.. ', Max);  
  C := ReadKey;  
End.
```


GetPalette procedure Graph

Declaration Procedure GetPalette(var Palette : PaletteType);

Function Returns the current palette, and its size.

Remark The PaletteType record is declared in the Graph unit as:

```
Type
  PaletteType = Record
    Size : Integer;
    Colors : Array[0..15] Of ShortInt;
  End;
```

Upon return from GetPalette, the Size field equals the number of colors in the current palette and Colors contains the actual colors.

See also GetDefaultPalette, SetAllPalette, SetPalette, SetRGBPalette

Example Program GetPalette_Demo;
Uses Graph;
Var

```
  Driver,Mode : Integer;
  Palette : PaletteType;
Begin
  Driver := DETECT;
  InitGraph(Driver,Mode,'');
  GetPalette(Palette);
  CloseGraph;
```

End.

GetPaletteSize function

Graph

Declaration Function GetPaletteSize : Integer;

Function Returns the number of entries in the palette.

Remark The range of the returned result depends on the mode, as follows:

Name	GetPaletteSize
CGAHi, MCGAHi, EGAMonoHi, PC3270Hi, StHigh	2
CGAC0, CGAC1, CGAC2, CGAC3, EGA64Hi, StMedium, AmigaWB	4
EGAHi	8
EGALo, VGAHi, StLow, AmigaWB16	16
AmigaLo32	32

See also SetPalette, SetAllPalette, SetRGBPalette

Example

```
Program GetPaletteSize_Demo;  
Uses Graph,Crt,UtilUnit;  
Var  
  Driver,Mode : Integer;  
  C : Char;  
Begin  
  Driver := DETECT;  
  InitGraph(Driver,Mode,'');  
  OutText(Int2Str(GetPaletteSize)+' colors in the  
palette.');
```

```
  C := ReadKey;  
  CloseGraph;  
End.
```

GetPixel function Graph

Declaration Function GetPixel(X,Y : Integer) : Word;

Function Returns the color of the specified pixel.

See also PutPixel, Graph

GetTextSettings procedure Graph

Declaration Procedure GetTextSettings(var TextInfo :
TextSettingsType);

Function Returns the current text settings.

Remark The TextSettingsType record is declared in the
Graph unit as:

Type
TextSettingsType= Record
 Font : Word;
 xDirection : Word;
 CharSize : Word;
 Horiz : Word;
 Vert : Word;
End;

Upon return from GetTextSettings, TextInfo will contain the current values as set by SetTextStyle and SetTextJustify:

Font is the text appearance (DefaultFont.. GothicFont).

Direction is the text direction (HorizDir, VertDir). Only horizontal text is supported on the Amiga.

CharSize is the size of text.

Horiz is the horizontal justification (LeftText..RightText). Vert is the vertical justification (BottomText..TopText).

See also *SetTextStyle, SetTextJustify*

Example

```
Program GetTextSettings_Demo;
Uses Graph,Crt,UtilUnit;
Var
  Driver,Mode,Th : Integer;
  C : Char;
  TextInfo : TextSettingsType;
Begin
  Driver := DETECT;
  InitGraph(Driver,Mode,'');
  Th := TextHeight('M');
  GetTextSettings(TextInfo);
  With TextInfo Do
    Begin
      OutTextXY(0,Th*1,'Current font      :
'+Int2Str(Font));
      OutTextXY(0,Th*2,'Current direction:
'+Int2Str(Direction));
      OutTextXY(0,Th*3,'Current charsize :
'+Int2Str(CharSize));
      OutTextXY(0,Th*4,'Horizontal just. :
'+Int2Str(Horiz));
      OutTextXY(0,Th*5,'Vertical just.:
'+Int2Str(Vert));
    End;
    C := ReadKey;
    CloseGraph;
  End.
```

GetFillSettings procedure Graph

Declaration Procedure GetFillSettings(var FillInfo : FillSettingsType);

Function Returns the current fill settings.

Remark The FillSettingsType record is declared in the Graph unit as:

```
Type
FillSettingsType= Record
Pattern : Word;
Color : Word;
End;
```

Upon return from GetFillSettings, FillInfo will contain the currently active fill pattern and fill color.

See also *GetFillPattern, SetFillStyle, SetFillPattern, FloodFill, FillEllipse, FillPoly, Bar, Bar3D, PieSlice*

Example Program GetFillSettings_Demo;
Uses Graph, Crt, UtilUnit;
Var
Driver, Mode : Integer;
C : Char;
FillInfo : FillSettingsType;
Begin
Driver := DETECT;
InitGraph(Driver, Mode, '');
GetFillSettings(FillInfo);
With FillInfo Do
Begin
OutTextXY(0, 0, 'Current fill pattern is :'
+ Int2Str(Pattern));
OutTextXY(0, TextHeight('M'),
'Current fill color is : ' + Int2Str(Color));
End;
C := ReadKey;
CloseGraph;
End.

GetTime procedure **DOS**

Declaration Procedure GetTime(var Hour, Min, Sec, Sec100 : Word);

Function Returns the current time according to the operating system.

Remark The ranges of the returned values are:

Hour	0..23
Min	0..59
Sec	0..59
Sec100	0..99

Note that the Sec100 parameter, will always return a multiple of 2 on the Amiga.

See also SetTime, GetDate

Example Program GetTime_Demo;

Uses DOS, Crt;

```
Var
  Hour, Minute, Second, Hundreds : Word;
Begin
  Repeat
    GetTime(Hour, Minute, Second, Hundreds);
    GoToXY(1, 1); ClrEol;
    Write(Hour, ': ', Minute, '. ', Second);
  Until keyPressed;
End.
```

GetViewSettings procedure Graph

Declaration Procedure GetViewSettings(var ViewPort : ViewPortType);

Function Returns information about the viewport.

Remark The ViewPortType record is declared in the Graph unit as:

Type

```
ViewPortType = Record  
  X1,Y1,X2,Y2 : Integer;  
  Clip : Boolean;  
End;
```

Upon return from GetViewSettings, ViewPort will contain the current viewport settings:

X1,Y1 specifies the upper-left corner of the viewport.

X2,Y2 specifies the lower-right corner of the viewport.

Clip specifies the clipping state of the viewport.

See also SetViewPort, ClearViewPort

Example Program GetViewSettings_Demo;
Uses Graph,Crt,UtilUnit;

```
Var  
  Th,  
  Driver,Mode : Integer;  
  C : Char;  
  ViewPortInfo: ViewPortType;  
Begin  
  Driver := DETECT;  
  InitGraph(Driver,Mode,'');  
  Th := TextHeight('M');  
  GetViewSettings(ViewPortInfo);  
  With ViewPortInfo Do  
  Begin
```

```

    OutTextXY(0,Th*0,'Minimum X co-ordinate :
'+Int2Str(X1));
    OutTextXY(0,Th*1,'Minimum Y co-ordinate :
'+Int2Str(Y1));
    OutTextXY(0,Th*2,'Maximum X co-ordinate :
'+Int2Str(X2));
    OutTextXY(0,Th*3,'Maximum Y co-ordinate :
'+Int2Str(Y2));
    MoveTo(0,Th*4);
    OutText('Clipping is currently: ');
    Case Clip Of
    True  : OutText('ON');
    False : OutText('OFF');
    End;
    End;
    C := ReadKey;
    CloseGraph;
End.

```

GetX function Graph

Declaration Function GetX : Integer;

Function Returns the X co-ordinate of the current pointer.

Remark The current pointer, also referred to as the CP, specifies the current position of the graphic cursor inside the current viewport.

See also *GetY, MoveTo, MoveRel, LineTo, LineRel, GetMaxX, GetMaxY*

Example

```

Program GetX_Demo;
Uses Graph,Crt;
Var
    Driver,Mode : Integer;
    C : Char;
Procedure CurrentPos;
Var
    Gx,Gy : String;
Begin
    Str(GetX,Gx);
    Str(GetY,Gy);
    OutText('This is position '+Gx+', '+Gy);
End;
Begin
    Driver := DETECT;

```



```

InitGraph(Driver,Mode,'');
MoveTo(10,10); CurrentPos;
MoveTo(100,100); CurrentPos;
MoveTo(200,200); CurrentPos;
C := ReadKey;
CloseGraph;
End.

```

GetY function Graph

Declaration Function GetY : Integer;

Function Returns the Y co-ordinate of the current pointer.

Remark The current pointer, also referred to as the CP, specifies the current position of the graphic cursor inside the current viewport.

See also *GetX, MoveTo, MoveRel, LineTo, LineRel, GetMaxX, GetMaxY*

Example

```

Program GetY_Demo;
Uses Graph,Crt;
Var
  Driver,Mode : Integer;
  C : Char;
Procedure CurrentPos;
Var
  Gx,Gy : String;
Begin
  Str(GetX,Gx);
  Str(GetY,Gy);
  OutText('This is position '+Gx+', '+Gy);
End;
Begin
  Driver := DETECT;
  InitGraph(Driver,Mode,'');
  MoveTo(10,10); CurrentPos;
  MoveTo(100,100); CurrentPos;
  MoveTo(200,200); CurrentPos;
  C := ReadKey;
  CloseGraph;
End.

```

GotoXY procedure Crt

Declaration Procedure GotoXY(Col, Lin : Integer);

Function Positions the cursor at specific screen co-ordinates.

Remark The top left of the window is (1,1). The range of co-ordinates that can be used depends on the size of the current CLI window which the user may change. You can find the maximum X and Y co-ordinates used the WindMax function as in the example below.

Beware that some programs written for the PC and Atari will assume that the screen is 80 by 25 characters.

See also ClrEos, InsLine, DelLine

Example Program GoToXY_Demo;

```
Var
  X,Y : Integer;
Begin
  For X := 1 to LO(WindMax) DO
    For Y := 1 to Hi(WindMax) DO
      Begin
        GoToXY(X,Y); Write(' ');
      End;
    End.
  End.
```

GraphDefaults procedure Graph

Declaration Procedure GraphDefaults;

Function Resets the Graph unit to its default values.

Remark GraphDefaults does the following:

Sets the viewport to the entire screen, with clipping on.

Restores the Graph unit default palette, and sets the foreground color to 1.

Sets the line style to be solid and the line width to be 1 pixel.

Sets the fill pattern to be solid, and the fill color to be the highest possible color.

Sets the text font to be the default font, the direction to be horizontal and the size to be 1.

See also *InitGraph*

GraphErrorMsg function

Graph

Declaration Function GraphErrorMsg(ErrorCode : Integer) : string;

Function Returns an error message corresponding to a Graph unit error code.

Remark GraphErrorMsg can be used together with the GraphResult function to obtain a string with an explanation to one of the Graph unit error codes.

Graph contains 19 predefined error messages:

0 No error

- 1 Graph not initialized. Use InitGraph.
- 2 Graphics hardware not detected.
- 3 Device driver not found.
- 4 Invalid device driver.
- 5 Not enough memory to load driver.
- 6 Out of memory in scan fill.
- 7 Out of memory in flood fill.
- 8 Font file not found.
- 9 Not enough memory to load font.
- 10 Invalid graphics mode for selected driver.
- 11 Graphics error.
- 12 Graphics I/O error.
- 13 Invalid font file.
- 14 Invalid font number.
- 15 Could not open graphics library
- 16 Could not open intuition
- 17 Could not open layers
- 18 Area operation failure

If the `ErrorCode` parameter lies outside the above range, the returned string will contain:

Graph error #xx

Otherwise the returned string will contain one of the above listed error messages.

See also `GraphResult`

Example

```
Program GraphErrorMsg_Demo;
Uses Graph,Crt;
Var
  Driver,Mode,X : Integer;
  C : Char;
Begin
  Driver := DETECT;
  InitGraph(Driver,Mode,'');
  For X := GrOK Downto GrInvalidFontNum Do
  Begin
    OutText(GraphErrorMsg(x));
    MoveTo(0,Succ(Abs(X))*TextHeight('M'));
  End;
  C := ReadKey;
  CloseGraph;
End.
```

GraphResult function

Graph

Declaration Function `GraphResult` : Integer;

Function Returns the error status of the last executed Graph command.

Remark The following error codes are declared in the Graph unit:

```
Const
  GrOK = 0;
  GrNoInitGraph = -1;
  GrNotDetected = -2;
  GrFileNotFound = -3;
  GrInvalidDriver = -4;
  GrNoLoadMem = -5;
  GrNoScanMem = -6;
```

GrNoFloodMem	=	-7;
GrFontNotFound	=	-8;
GrNoFontMem	=	-9;
GrInvalidMode	=	-10;
GrError	=	-11;
GrIOError	=	-12;
GrInvalidFont	=	-13;
GrInvalidFontNum	=	-14;
GrNoGraphLib	=	-15;
GrNoIntuiLib	=	-16;
GrNoLayersLib	=	-17;
GrAreaFail	=	-18;

Detailed descriptions of these error messages can be found under `GraphErrorMessage` above.

The following procedures update `GraphResult`:

DetectGraph	DrawPoly	FillPoly
SetActivePage	SetAllPalette	SetFillPattern
SetFillStyle	SetGraphMode	SetLineStyle
SetPalette	SetTextJustify	SetTextStyle
SetViewPort	SetVisualPage	

Note that `GraphResult` is cleared every time it is called, just like the `IOresult` function.

See also

`GraphErrorMsg`

Example

```

Program GraphResult_Demo;
Uses Graph,Crt;
Var
  Driver,Mode : Integer;
  C : Char;
Begin
  Driver := DETECT;
  InitGraph(Driver,Mode,'');
  SetActivePage(22222);
  If GraphResult = GrOK then
    OutText('Impossible!')
  Else
    OutText('Page not available.');
```

C := ReadKey;
 CloseGraph;
 End.

Halt procedure

Declaration Procedure Halt [(ExitCode : Integer) ;

Function Halts the current program.

Remark If the optional parameter ExitCode is not specified, an exitcode of 0 will be used.

See also Exit

Example

```
Program Halt_Demo;
Begin
  Writeln('Entering step 1. ');
  Begin
    Writeln('Entering step 2. ');
    Begin
      Halt(1);
    End;
    Writeln('This will not be executed. ');
  End;
  Writeln('Neither will this. ');
End.
```

Hi function

Declaration Function Hi(N : Integer) : Byte;

Function Returns the high byte of N.

See also Lo, LoWord, HiWord

Example

```
Program Hi_Demo;
Var
  X : Byte;
Begin
  X := Hi($1234); { = $12 }
End.
```

HighVideo procedure Crt

Declaration Procedure HighVideo;

Function Switch to high intensity video.

Remark This is implemented using bold on the Amiga as this is the nearest effect available.

See also NormVideo, TextBackground, TextColor

Example

```
program HighVideo_Test;
Uses CRT;
var i:integer;
begin
  HighVideo;
  Writeln('This is high intensity ');
  NormVideo;
  Writeln('This is normal again');
End.
```

HiWord function

Declaration Function HiWord(N : LongInt) : Integer;

Function Returns the high word of N.

See also LoWord, Hi, Lo

Example

```
Program HiWord_Demo;

Var
  X : Integer;
Begin
  X := HiWord($12345678); { = $1234 }
End.
```

ImageSize procedure Graph

Declaration Function ImageSize(x1, y1, x2, y2 : Integer) : LongInt;

Function Returns the size, i bytes, of a specified screen area.

Remark ImageSize calculates the storage needed to save a rectangular area of the screen, using the GetImage procedure.

The point (x1,y1) is the upper-left, and (x2,y2) is the lower-right corner of the screen area. These are included for compatibility with the PC and Atari versions of this call. ImageSize actually returns the size of a data structure that is used internally by the system. Thus it is essential that you use ImageSize to find the amount of memory required.

See also GetImage, PutImage, FreeImage, ImageSize, Graph

Inc procedure

Declaration Procedure Inc(N [; Count]);

Function Increments an ordinal variable by 1 or by Count.

Remark N and the optional parameter Count are both simple types. If Count is not specified N will be incremented by 1. Otherwise N will be incremented by Count.

See also Dec, Pred

Example

```
Program Inc_Demo;
Var
  L : LongInt;
Begin
  L := 0;
  Repeat
    Inc(L); { Increase L by 1 }
    Inc(L,2); { Increase L by 2 }
  Until (L > 100000);
End.
```


Include function

- Declaration** Function Include(NewStr, MainStr : String; Index : Byte);
- Function** Returns MainStr with NewStr inserted at position Index.
- Remark** The rules for the Include function are the same as for the Insert procedure.
- See also** Insert
- Example** Program Include_Demo;
- ```
Var
 Res : String;
Begin
 Res := Include('IS ', 'AMIGA GREAT', 7);
 Writeln(Res);
End.
```

# InitGraph procedure Graph

---

- Declaration** Procedure InitGraph(var GraphDriver, GraphMode: Integer; Path: string);
- Function** Initializes and starts up the Graph unit.
- Remark** GraphDriver specifies the graphics driver to use. The only supported drivers on the Amiga are declared in the Graph unit as:
- ```
Const
  DETECT = 0;
  Amiga = 14;
```
- GraphMode specifies the mode of the selected driver.

The legal modes are declared in the Graph unit as:

```
Const
  CGAC0 = 0
  CGAC1 = 1
  CGAC2 = 2
  CGAC3 = 3
  CGAHi = 4
  MCGAHi = 5
  EGALo = 6
  EGAHi = 7
  EGA64Hi = 8
  EGAMonoHi = 9
  VGAHi = 10
  PC3270Hi = 11
  StLow = 12
  StMedium = 13
  StHigh = 14
  AmigaWB = 15
  AmigaWB16 = 16
  AmigaLo32 = 17
  AmigaCustom = 18
```

Path is ignored by InitGraph.

There are two different ways of initializing the Graph unit:

Auto detection:

When using auto detection, InitGraph will automatically find out what driver and mode to use. Auto detection is activated when the GraphDriver parameter equals DETECT.

User selection:

User selection is activated if GraphDriver does not equal DETECT.

When user selection has been requested, it is imperative that a legal driver (i.e. Amiga) and graphics mode, as in the table above, are passed to InitGraph through the GraphDriver and GraphMode parameters.

If `InitGraph` finds that the chosen driver does not match your monitor type, or that the chosen driver and mode cannot be used together, the program will halt, and an error message will be displayed.

See also *DetectGraph, CloseGraph, GetModeRange, SetGraphMode, GraphResult, GraphErrorMsg, GraphDefaults*

Example

```
Program InitGraph_Demo;
Uses Graph,Crt,UtilUnit;
Var
  Driver,Mode,Th : Integer;
  C : Char;
Begin
  { Auto detection. }
  Driver := DETECT;
  InitGraph(Driver,Mode,'');
  Th := TextHeight('M');
  OutTextXY(0,0,'The detected driver is:
'+Int2Str(Driver));
  OutTextXY(0,Th,'The detected mode is :
'+Int2Str(Mode));
  C := ReadKey;
  CloseGraph;
  { User selection. }
  Driver := Amiga;
  Mode := VgaHi;
  InitGraph(Driver,Mode,'');
  OutTextXY(0,0,'The selected driver is:
'+Int2Str(Driver));
  OutTextXY(0,Th,'The selected mode is :
'+Int2Str(Mode));
  C := ReadKey;
  CloseGraph;
End.
```

Insert procedure

- Declaration** Procedure Insert(NewStr : String; var MainStr : String; Index : Byte);
- Function** Inserts NewStr into MainStr at position Index.
- Remark** If Index is greater than the length of MainStr, nothing will be inserted.
- See also** *Include, Copy, Delete, Length, Pos*
- Example** Program Insert_Demo;

```
Var
  S : String;
Begin
  S := 'Something missing';
  Writeln('Before calling insert: ',S);
  Insert(' is',S,Pos(' ',S));
  Writeln('After calling insert : ',S);
End.
```

InsLine procedure CRT

- Declaration** Procedure InsLine;
- Function** Inserts a blank line on the screen.
- Remark** The position at which the blank line is inserted, depends on the current cursor location. All lines under the current line will be scrolled down, and the bottom line is lost.
- See also** *DellLine, GotoXY*
- Example** Program InsLine_Demo;

```
Uses Crt;
Var
  X,Y : Integer;
Begin
  For X := 1 to Lo(WubdMax) DO
    For Y := 1 to Hi(WindMax) DO
      Begin
```

```

        GoToXY(X,Y); Write(Random(2));
    End;
    GoToXY(30,10);
    Writeln('Press ENTER to go on');
    Readln;
    GoToXY(1,1);
    For Y := 1 to Hi(WindMax) DO
        Inslne;
    End.

```

Int function

Declaration Function Int(N : Real) : Real;

Function Returns the integer part of N.

Remark Trunc, Round, Abs

Example Program Int_Demo;

```

Var
    R : Real;
Begin
    R := INT(1234.5678); { R = 1234.00 }
End.

```

IOresult function

Declaration Function IOresult : Integer;

Function Returns the error status of the last IO operation.

Remark It is important to notice that **IOresult** is used as an error status for several IO routines in the system. Therefore it is not reliable to write a program like this:

```

{ Do some file operations here. }
Writeln('Error code: ',IOresult);

```

Since Writeln also uses **IOresult** to report errors. (Thereby setting **IOresult** to 0 when writing 'Error code: ').

Instead use something like this:

```
{ Do some file operations here. }  
MyVar := IOresult;  
Writeln('Error code: ',MyVar);
```

Example

Program IOresult_Demo;

```
Var  
  F : FILE;  
  I : Integer;  
Begin  
  {$I-} Reset(F, 'NOTHING.#{?}'); {$I+}  
  I := IOresult; { Always make a copy. }  
  Writeln('IOresult says: ',I);  
End.
```

KeyPressed function CRT

Declaration Function KeyPressed : Boolean;

Function Checks to see if the keyboard has been pressed.

Remark KeyPressed will return TRUE if a key is waiting to be read, otherwise FALSE.

See also ReadKey

Example Program KeyPressed_Demo;

```
Uses Crt;  
Begin  
  Repeat  
    Writeln('Please press a key.');
```

Until KeyPressed;
 Writeln('Thank you.');

```
End.
```

Length function

Declaration Function Length(S : String) : Byte;

Function Returns the length of string S.

Remark The returned value will be in the range of 0..255.

See also Copy, Concat, Insert, Include, Delete, Omit, Pos

Example

```
Program Length_Demo;
Var
  S : String;
Begin
  Write('Enter a string: '); Readln(S);
  Writeln;
  Writeln('You entered ',Length(S),' characters');
  Writeln('You entered ',Ord(S[0]),' characters');
End.
```

Line procedure Graph

Declaration Procedure Line(x1, y1, x2, y2 : Integer);

Function Draws a line from point (x1,y1) to point (x2,y2).

Remark The line is drawn in the current color and line pattern, using the write mode defined by the SetWriteMode procedure.

See also LineTo, LineRel, SetWriteMode, SetLineStyle, GetLineSettings

Example

```
Program Line_Demo;
Uses Graph,Crt;
Var
  Driver,Mode : Integer;
Begin
  Driver := DETECT;
  InitGraph(Driver,Mode,'');
  Repeat
    SetColor(Random(GetMaxColor)+1);
    Line(0,0,Random(GetMaxX),Random(GetMaxY));
  Until KeyPressed;
  CloseGraph;
End.
```

LineRel procedure Graph

Declaration Procedure LineRel(Dx, Dy : Integer);

Function Draws a line from the CP to a point relative to its current location.

Remark The line is drawn in the current color and line pattern, using the write mode defined by the SetWriteMode procedure.

See also LineTo, Line, SetWriteMode, SetLineStyle, GetLineSettings

Example

```
Program LineRel_Demo;
Uses Graph,Crt;
Var
  Driver,Mode : Integer;
Begin
  Driver := DETECT;
  InitGraph(Driver,Mode,'');
  MoveTo(GetMaxX DIV 2,GetMaxY DIV 2);
  Randomize;
  Repeat
    SetColor(Random(GetMaxColor)+1);
    LineRel(Random(3)-1,Random(3)-1);
  Until KeyPressed;
  CloseGraph;
End.
```

LineTo procedure Graph

Declaration Procedure LineTo(X,Y : Integer);

Function Draws a line from the current pointer (CP) to point (X,Y).

Remark The line is drawn in the current color and line pattern, using the write mode defined by the SetWriteMode procedure.

See also LineRel, Line, SetWriteMode, SetLineStyle, GetLineSettings, Graph

Example

```
Program LineTo_Demo;  
Uses Graph,Crt;  
Var  
  Driver,Mode : Integer;  
Begin  
  Driver := DETECT;  
  InitGraph(Driver,Mode,'');  
  Repeat  
    SetColor(Random(GetMaxColor)+1);  
    LineTo(Random(GetMaxX),Random(GetMaxY));  
    Delay(500);  
  Until KeyPressed;  
  CloseGraph;  
End.
```

Ln function

Declaration Function (N) : Real;

Function Returns the natural logarithm of N.

Remark N is an integer-type or real-type expression. The result is the natural logarithm of N.

See also Exp, ValidReal

Example Program Ln_Demo;

```
Var  
  R : Real;  
Begin  
  R := LN(100);  
End.
```

Lo function

Declaration Function Lo(N : Integer) : Byte;

Function Returns the low byte of N.

Remark Hi, HiWord, LoWord

Example Program Lo_Demo;

```
Var
  X : Byte;
Begin
  X := Lo($1234); { = $34 }
End.
```

LockAlertWindow function DOS

Declaration Function LockAlertWindow(OldPointer: Pointer): Pointer;

Function Used to suppress and re-enable disk insertion and error alerts.

Remark If these alerts are currently enabled, i.e. the current value of the pr_WindowPtr field of this task is -1. this field will be set to -1, thus causing disk insertion requests etc. to be returned to the program as an AmigaDos error. This function returns the previous value of the pr_WindowPtr.

If the alerts were already disabled they will be re-enabled using the OldPointer parameter as the new value for this field This will have been obtained via a previous call to this function.

Note that this function provides a strict toggle; therefore such calls may not be nested.

See also FExpandLock, FSplit, FindFirst

Example

```
Program LockAlertWindow_Demo;
Uses DOS;
VAR p:Pointer;
    f:Text;
    status:Integer;
begin
    P:=LockAlertWindow(NIL); {Disable requesters }
    {$I-} rewrite(f,'rad:temp'); {$I+ }
    P:=LockAlertWindow(P);      {Restore WindowPtr}
    status:=IoResult;

    If status<>0 then begin
        writeln('can''t open rad:temp');
        halt;
        end;
    { Now we can use f}
End.
```

LoWord function

Declaration Function LoWord(N : LongInt) : Integer;

Function Returns the low word of N.

Remark HiWord, Hi, Lo

Example Program Lo_Word;

```
Var
    X : Integer;
Begin
    X := LoWord($12345678); { = $5678 }
End.
```

MaxAvail function

Declaration Function MaxAvail : LongInt;

Function Returns the size of the largest contiguous memory block in the heap.

Remark *MemAvail, New, GetMem*

Example Program MaxAvail_Demo;

```
Var
  L : LongInt;
Begin
  L := MaxAvail;
  Writeln('Size of the biggest hole in the heap is: ',L);
End.
```

MemAvail function

Declaration Function MemAvail : LongInt;

Function Returns the sum of the sizes of all free memory blocks in the heap.

See also *MaxAvail*

Example Program MemAvail_Demo;

```
Var
  L : LongInt;
Begin
  L := MemAvail;
  Writeln('Bytes free in the heap is: ',L);
End.
```

MkDir procedure DOS

Declaration Procedure MkDir(Dir : DirStr);

Function Creates a new directory.

Remark DirStr is declared in the DOS unit:

```
Type  
  DirStr = String[127];
```

See also ChDir, RmDir, GetDir

Example Program Make_Directory;

Uses DOS;

```
Begin  
  MkDir(ParamStr(1));  
End.
```

Move procedure

Declaration Procedure Move(var Source, Dest; Count : LongInt);

Function Move Count bytes from Source to Dest.

Remark Source and Dest can be any variable. Count specifies how many bytes to move.

If Count is greater than the size of Dest, other code or data may be overwritten, causing unpredictable problems. Therefore it is a good idea to use the SizeOf function to specify how many bytes to move:

```

Var
  Big : LongInt;
  Small : Integer;
Begin
  Move(Big,Small,4);
  { The very dangerous way of doing it! }
  Move(Big,Small,SizeOf(Small));
  { The safe way of doing it. }
End.

```

See also *Sizeof, FillChar*

Example Program Move_Demo;
 Var
 Big1,
 Big2 : Packed Array[1..10000] OF Integer;
 Begin
 Move(Big1,Big2,SizeOf(Big2));
 End.

MoveRel procedure Graph

Declaration Procedure MoveRel(Dx, Dy : Integer);

Function Moves the current pointer to a point relative to its current location.

Remark The current pointer can be moved backwards by using a negative value for Dx, and upwards by using a negative value for Dy.

See also *MoveTo, GetX, GetY, LineRel, LineTo, Graph*

Example Program MoveRel_Demo;
 Uses Graph,Crt;
 Var
 Driver,Mode : Integer;
 Begin
 Driver := DETECT;
 InitGraph(Driver,Mode,'');
 Repeat
 MoveRel(Random(5)-2,Random(3));
 PutPixel(GetX,GetY,Random(GetMaxColor+1));
 If (GetY >= GetMaxY) then
 MoveTo(Random(GetMaxX),0);
 Until KeyPressed;
 CloseGraph;
 End.

MoveTo procedure Graph

Declaration Procedure MoveTo(X, Y : Integer);

Function Moves the current pointer to a new position.

See also MoveRel, GetX, GetY, LineTo, LineRel

Example

```
Program MoveTo_Demo;
Uses Graph,Crt;
Var
  Driver,Mode : Integer;
  C : Char;
Begin
  Driver := DETECT;
  InitGraph(Driver,Mode,'');
  MoveTo(GetMaxX,GetMaxY);
  LineTo(0,0);
  C := ReadKey;
  CloseGraph;
End.
```

New procedure

Declaration Procedure New(var P : Pointer);

Function Creates a new dynamic variable and sets P to point to it.

Remark P is a pointer variable of any type. The size of the allocated memory block corresponds to the size of the type that P points to.

See also Dispose, GetMem, FreeMem

Example Program New_Demo;

Type

```
PersonType = RECORD
  Name : String[20];
  Age : Byte;
End;
```

Var

```
PersonData : ^PersonType;
```

Begin

```
New(PersonData); { Allocate space in heap }
With PersonData^ DO
```

```
Begin
```

```
Write('Enter name: '); ReadLn(Name);
```

```
Write('Enter age : '); ReadLn(Age);
```

```
End;
```

```
Dispose(PersonData); { Dispose the space }
```

```
End.
```


NormVideo procedure Crt

Declaration Procedure NormVideo;

Function Switch to normal intensity video, if high intensity video has been used.

Remark This cancels the effect of the HighVideo procedure.

See also HighVideo, TextBackground, TextColor

Example

```
program NormVideo_Test;
Uses CRT;
var i:integer;
begin
  HighVideo;
  Writeln('This is high intensity ');
  NormVideo;
  Writeln('This is normal again');
End.
```

Odd function

Declaration Function Odd(N) : Boolean;

Function Tests to see if the N parameter is an odd number.

Remark N is an integer type value.

Example Program Odd_Demo;

```
Var
  L : LongInt;
Begin
  Repeat
    Write('Enter a number: '); Readln(L);
    If Odd(L) then
      Writeln('ODD')
    Else
      Writeln('EVEN');
  Until (L = 0);
End.
```

Omit function

- Declaration** Function Omit(S : String; Index, Count : Integer) : String;
- Function** Returns S with Count characters deleted, starting at position Index.
- Remark** The rules for the Omit function are the same as for the Delete procedure.
- See also** Delete
- Example** Program Omit_demo;
- ```
Var
 Res : String;
Begin
 Res := Omit('AMIGA IS NOT GREAT',10,4);
 Writeln(Res);
End.
```

# Ord function

---

- Declaration** Function Ord( x ) : (Integer type)
- Function** Returns the ordinal number of an ordinal-type value.
- Remark** Depending upon the x parameter, the result returned by Ord will be one of these types: Byte, Integer or LongInt.
- See also** Ord4, Chr
- Example** Program Ord\_Demo;
- ```
Var
  C : Char;
Begin
  Write('Enter something: ');
  C := ReadKey;
  Writeln;
  Writeln('The ordinal value of ',C,' is ',ORD(C));
End.
```

Ord4 function

Declaration Function Ord4(x) : LongInt;

Function Returns the ordinal number of an ordinal-type value.

Remark This function does the same as the Ord function, but the result is always a long integer.

See also Ord, Chr

OutText procedure Graph

Declaration Procedure OutText(TextString : string);

Function Outputs a string of text at the current pointer (CP).

Remark TextString is written at the current pointer, using the current text style, direction and justification.

If the direction is HorizDir and the justification is LeftText, the X part of the current pointer will be updated according to the length of the output string.

See also OutTextXY, TextHeight, TextWidth, SetTextStyle, SetTextJustify

Example

```
Program OutText_Demo;
Uses Graph,Crt,UtilUnit;
Var
  Driver,Mode : Integer;
Begin
  Driver := DETECT;
  InitGraph(Driver,Mode,'');
  Repeat
    MoveTo(Random(GetMaxX),Random(GetMaxY));
    OutText('(' + Int2Str(GetX) + ',' + Int2Str(GetY) + ')');
    Until KeyPressed;
  CloseGraph;
End.
```

OutTextXY procedure Graph

- Declaration** Procedure OutTextXY(X,Y : Integer;
TextString : string);
- Function** Outputs a string of text at the specified co-ordinates.
- Remark** (X,Y) is the point where the text is written.
TextString is the string to write.
- The current pointer is not updated by a call to OutTextXY.
- See also** OutText, TextHeight, TextWidth, SetTextStyle,
SetTextJustify
- Example** Program OutTextXY_Demo;
Uses Graph,Crt,UtilUnit;
Var
Driver,Mode,P : Integer;
C : Char;
Begin
Driver := DETECT;
InitGraph(Driver,Mode,'');
P := 0;
Repeat
OutTextXY(P,P,(' '+Int2Str(P)+' ',' '+Int2Str(P)+' '));
Inc(P,TextHeight('X'));
Until (P > GetMaxY) Or (P > GetMaxX);
C := ReadKey;
CloseGraph;
End.

PackTime procedure DOS

- Declaration** Procedure PackTime(D : DateTime; var
Result : PackedTime);
- Function** Packs a 16-byte DateTime record into a PackedTime structure.
- Remark** The DateTime record is declared in the DOS unit:

```

DateTime      = RECORD
  DayOfWeek   : Word;
  Year        : Word;
  Month       : Word;
  Day         : Word;
  Hour        : Word;
  Min         : Word;
  Sec         : Word;
  Sec100      : Word;
END;

```

PackedTime corresponds to the type `tDateStamp` in the AmigaDOS unit and is used in calls to `SetFTime`, `GetFTime` and `UnPackTime` procedures.

Note that on the Atari and under MS-DOS a `LongInt` is used rather than `PackedTime`.

See also

UnpackTime, SetFTime

Example

```

Program PackTime_Demo;
Uses DOS;
Var
  U : DateTime; { The unpacked time and date }
  P : PackedTime; { The packed time and date }

Procedure WriteData;
Begin
  With U DO
    Begin
      WriteLn('Time: ',Hour,'.',Min,',',Sec);
      WriteLn('Date: ',Day,'-',Month,'/',Year);
    End;
  Writeln;
End;

Begin { main }
  With U DO
    Begin
      Year := 1990;
      Month := 12;
      Day := 24;
      Hour := 12;
      Min := 0;
      Sec := 0;
    End;
  Writeln('Before packing:'); WriteData;
  PackTime(U,P);
  UnPackTime(P,U);
  Writeln('After unpacking:'); WriteData;
End.

```

Page procedure

Declaration Procedure Page [(var F : text)] ;

Function Writes a formfeed character to file F.

Remark If F is omitted, the standard file **Output** is assumed.
Page is equivalent to:

```
Write( F, Chr(12) );
```

Example Program Page_Demo;

```
Var
  T : Text;
Begin
  Rewrite(T,'printer.list');
  Writeln(T,'First page.');
```

See also

```
  Page(T);
  Writeln(T,'Second page.');
```

Example

```
  Page(T);
  Writeln(T,'Third page.');
```

Uses DOS;

```
  Page(T);
  Close(T);
End.
```

PackTime procedure

Declaration Procedure PackTime(S : ardateTime; var
Result : PackTime) ;

Function Packs a 16-byte DateTime into a 10-byte PackTime
structure.

Remark The DateTime is given in the DOS units

ParamCount function DOS

Declaration Function ParamCount : Integer;

Function Returns the number of parameters that were passed to the program on the command line.

Remark Use the ParamStr function to obtain the value of the parameters.

See also ParamStr

Example Program ParamCount_Demo;

Uses DOS;

Var

Res, Count : Integer;

Begin

Res := ParamCount;

Writeln('You passed ',Res,' parameters to this program.');

Writeln;

If (Res = 0) then

HALT;

Writeln('Here is a list of the parameters:');

Writeln;

For Count := 1 to Res DO

Writeln('Parameter #',Count:2,' :=

',ParamStr(Count));

End.

ParamStr function DOS

Declaration Function ParamStr(Index : Integer) : String;

Function Returns command line parameter number Index.

Remark If Index is less than zero or greater than ParamCount, ParamStr will return an empty string. If Index is zero then the name of the program being run is returned.

See also ParamCount

Example Program ParamStr_Demo; { Displays 1 or more text files. }

Uses DOS;

Var

 DataFile : TEXT;

 Data : String;

 Count : Integer;

Procedure DisplayFile(Number : Integer);

Begin

 {\$I-} Reset(DataFile, ParamStr(Number)); {\$I+}

 If (IOresult <> 0) then

 Begin

 Writeln('Error while opening file:
' ,ParamStr(Number));

 EXIT;

 End;

 While NOT(Eof(DataFile)) DO

 Begin

 Readln(DataFile, Data);

 Writeln(Data);

 End;

 Close(DataFile);

End; { DisplayFile }

Begin

 For Count := 1 to ParamCount DO

 DisplayFile(Count);

End.

Pi constant

Declaration Const Pi = 3.14159265358979324;

Function Returns the value of Pi.

Example Program Pi_Demo;

```
Begin
  Writeln('Pi is: ',Pi :23:20);
End.
```

PieSlice procedure Graph

Declaration Procedure PieSlice(X, Y : Integer;
StAngle, EndAngle, Radius : Word);

Function Draws and fills a pie slice.

Remark (X,Y) specifies the centre point, StAngle the start angle, EndAngle the end angle and Radius the radius of the pie slice.

The pie slice is drawn in the current color, and filled using the current fill settings.

The shape of the pie slice is affected by the current aspect ratio settings.

See also SetAspectRatio, SetFillStyle, Arc, Circle, Rectangle, Sector, DrawPoly, Ellipse, Bar

Example Program PieSlice_Demo;
Uses Graph,Crt;
Var

```
  Slice,F,Hy,
  Driver,Mode : Integer;
  C : Char;
Begin
  Driver := DETECT;
  InitGraph(Driver,Mode,'');
  F := 360 DIV 12;
  Hy := GetMaxY DIV 2;
  For Slice := EmptyFill to CloseDotFill Do
  Begin
```

```

SetFillStyle(Slice,Random(GetMaxColor)+1);
SetColor(Random(GetMaxColor)+1);
PieSlice(GetMaxX DIV
2,Hy,(Slice*F)+2,(Succ(Slice)*F)-2,Hy);
End;
C := ReadKey;
CloseGraph;
End.

```

Pos function

Declaration Function Pos(SubStr, MainStr : String)
: Byte;

Function Searches MainStr for the first occurrence of SubStr.

Remark If SubStr does not occur in MainStr, the Pos function will return 0, else Pos will return the position in MainStr where SubStr was found.

Here are some examples:

```

Pos('HELLO','HELLO WORLD') = 1
Pos('CAL','PASCAL')         = 4
Pos('TS','COMMODORE-AMIGA') = 0

```

See also Copy, Insert, Delete, Length

Example Program Pos_Demo;

```

Var
  MainStr,
  SubStr : String;
  P : Byte;

Begin
  Write('Enter main string: ');
  Readln(MainStr);
  Write('Find what : '); Readln(SubStr);
  Writeln;
  P := Pos(SubStr,MainStr);
  If (P = 0) then
    Writeln('Not found.')
  Else
    Writeln('Found at position ',P);
End.

```

Pred function

Declaration Function Pred(X) : (Same type as parameter)

Function Returns the predecessor of the parameter.

Remark X is an ordinal-type value.

See also Succ, Dec, Inc

Example Program Pred_Demo;

```
Begin
  Writeln(Pred(2)); { = 1 }
End.
```

Ptr function

Declaration Function Ptr(Address : LongInt) : Pointer;

Function Converts a long integer value to a pointer type value.

Remark The result of **Ptr** is a pointer that points to the memory location specified by the value of **Address**.

See also SPtr, Addr

Example Program Ptr_Demo;

```
Uses DOS,Exec;

Var
  SysBasePtr: pExecBase;
  P : ^integer;
  I : Integer;
Begin
  SysBasePtr:= Ptr(4);
  Writeln('This is version SysBasePtr^.SoftVer);
End.
```

PutImage procedure Graph

Declaration Procedure PutImage(x, y, x2, y2 : Integer; var BitMap; BitBlt: Word);

Function Displays a stored image on the screen.

Remark BitMap contains the image to display; this is normally obtained via GetImage. BitBlt specifies the binary operation to use when putting the bits on the screen. The valid operations are

Const		
NormalPut	=	0; { Replace }
CopyPut	=	0; { Replace }
XorPut	=	1; { XOR }
OrPut	=	2; { OR }
AndPut	=	3; { AND }
NotPut	=	4; { NOT }

NormalPut and CopyPut will simply put the stored image on the screen. XORput will exclusive OR and AndPut will And the stored image with the existing screen contents. NotPut will NOT (invert) the stored image onto the screen.

See also ImageSize, GetImage, FreeImage

PutPixelprocedure Graph

Declaration Procedure PutPixel(X, Y : Integer; Pixel : Word);

Function Plots a pixel in the specified color at the specified point.

See also *GetPixel, GetColor*

Example

```
Program Pixel_Demo;
Uses Graph,Crt;
Var
  Rx,Ry,
  Driver,Mode : Integer;
Begin
  Driver := DETECT;
  InitGraph(Driver,Mode,'');
  Randomize;
  SetFillStyle(LtBkSlashFill,Random(GetMaxColor)+1);
  Bar3D(0,0,100,100,0,TopOff);
  Repeat
    Rx := Random(101);
    Ry := Random(101);
    PutPixel(100+Rx,100+Ry,GetPixel(Rx,Ry));
  Until KeyPressed;
  CloseGraph;
End.
```

RandomSeed variable

Declaration Var RandomSeed : Integer;

Function RandomSeed : Integer;

Remark Setting RandomSeed to a specific value can make the Random function return the same random numbers over and over again. This feature is useful in cryptography and simulation algorithms.

Random function

Declaration Function Random [(Max : LongInt) :
LongInt] | [: Real] ;

Function Returns a random number.

Remark If the optional Max parameter is specified, the result of Random will be a long integer in the range of 0..Max-1.

If no parameter is given to the Random function a random real number, in the interval $0 \leq x < 1$, will be returned.

See also Randomize, RandSeed

Example Program Random_Demo;

```
Var
  Tries ,
  Secret,
  Guess : Integer;

Begin
  Randomize;
  Writeln('Guess a number. '); Writeln;
  Secret := Random(100);
  Tries := 0;
  Repeat
    Inc(Tries);
    Write('Enter your guess number ',Tries,': ');
    Readln(Guess);
    If (Guess < Secret) then
      Writeln('Too low.')
    Else If (Guess > Secret) then
      Writeln('Too high. ');
  Until (Guess = Secret);
  Writeln('You guessed it in ',Tries,' attempts. ');
End.
```

Randomize procedure

Declaration Procedure Randomize;

Function Initializes the random-number generator with a random value.

Remark Every time a program is started, the random-number generator is initialized with the same value (0). The effect of this is that random numbers generated by a program will be the same every time the program is run.

There are 2 ways of getting around this problem:

Call `Randomize` to re-initialize the random-number generator.

Change the value of the `RandSeed` variable.

See also *Random, RandSeed*

Example Program Randomize_Demo;

```
Begin
  { This will be the same everytime the program is
  run. }
  Writeln(Random(1000));
  Randomize;
  { This will not! }
  Writeln(Random(1000));
End.
```

RandSeed variable

Declaration Var RandSeed : LongInt;

Function RandSeed is used as the seed for the random-number generator.

Remark Setting RandSeed to a specific value can make the Random function return the same random numbers over and over again. This feature is useful in encryption and simulation algorithms.

See also*Random, Randomize***Example**

Program RandSeed_Demo;

```

Begin
  RandSeed := 1234;
  WriteLn(Random(9999)); { This }
  RandSeed := 1234;
  WriteLn(Random(9999)); { and this will always be
the same. }
End.

```

Read procedure

Declaration Procedure Read [([var F;] v1
[,v2,..vn])];

Function Reads 1 or more components from file F into 1 or more variables.

Remark Read for text files:

F is a file variable of type text. If omitted the standard file **Input** is assumed. Each of the variables (v1,v2..vn) must be of type char, integer, real or string.

Read for typed files:

F is a file variable of any type except text. Each of the variables (v1,v2,vn) must be of the same type as the component type of file F.

See also *ReadLn, Write, WriteLn*

Example Program Read_Demo;

Uses DOS;

Var

T : Text;

C : Char;

Begin

{ \$I- } Reset(T,ParamStr(1)); { \$I+ }

If (IOresult = 0) then

Begin


```

While NOT(Eof(T)) DO
  Begin
    Read(T,C);
    Write(C);
  End;
  Close(T);
End
Else
  Writeln('File not found.');
```

End.

ReadKey function CRT

Declaration Function ReadKey : Char;

Function Reads one character from the keyboard.

Remark If no character is waiting to be read (see KeyPressed), a key must be pressed in order to exit the ReadKey function. Otherwise a character is read from the system's internal keyboard buffer.

See also KeyPressed, UpCase

Example

```

Program ReadKey_Demo;
Uses Crt;
Begin
  Repeat
    Writeln('Press SPACEBAR to stop this.');
```

Until (ReadKey = #32);

End.

ReadLn procedure

Declaration Procedure ReadLn [([var F : text;] v1 [,v2,..vn])];

Function Executes the Read procedure, and then skips to the next line in the file.

Remark If no next line can be found in the file, the Eof(F) will return true. If ReadLn is called with only a file variable as parameter, the file pointer will advance to the beginning of the next line in the file.

See also
Example

Read, WriteLn, Write

Program ReadLn_Demo;

Var

S : String;

Begin

Write('What is your name: ');

ReadLn(s);

End.

Rectangle procedure Graph

Declaration Procedure Rectangle(x1, y1, x2, y2 : Integer);

Function Draws a rectangle.

Remark (x1,y1) specifies the upper-left and (x2,y2) the lower-right corner of the rectangle.

The rectangle is drawn in the current color, using the current line style.

See also *SetWriteMode, Bar, Arc, Circle, PieSlice, Sector, DrawPoly, Ellipse*

Example

Program Rectangle_Demo;

Uses Graph,Crt;

Var

Driver,Mode : Integer;

Begin

Driver := DETECT;

InitGraph(Driver,Mode,'');

Repeat

SetColor(Random(GetMaxColor)+1);

Rectangle(Random(GetMaxX),Random(GetMaxY),

Random(GetMaxX),Random(GetMaxY));

Until KeyPressed;

CloseGraph;

End.

Rename procedure

Declaration Procedure Rename(OldName, NewName : String);

Function Renames the file OldName to NewName.

See also Erase

Example Program Rename_Demo;

Uses DOS;

Var

Current,

New : String;

Begin

Write('Enter current file name: ');

Readln(Current);

Write('Enter new file name : ');

Readln(New);

Rename(Current, New);

End.

Reset procedure

Declaration Procedure Reset(var F [; FileName:String]);

Function Opens or resets a file.

Remark F is a file variable of any type. FileName specifies the name of the file on the disk. The optional BufSize can be specified with files of type text to determine the size of a buffer.

The Reset procedure can be used in 2 different ways:

Reset(MyFile, 'NAME.EXT');

Opens the file MyFile. If the file is of type text it is opened as Read Only. Note that this form is not supported by Turbo Pascal.

Reset(MyFile);

Opens the file using the name that was previously specified using **Assign**. If the file is already open it positions the file pointer at the start of **MyFile**.

See also *Assign, ReWrite, Append, SetTextBuf*

Example Program **Reset_Demo**;

```
Uses DOS;
Const BufSize=1024;
Var
  TextF : Text;
  Data : String;
  Buffer : Packed Array[1..BufSize] of Char;
Begin
  {$I-}
  Assign(TextF,ParamStr(1));
  SetTextBuf(TextF,Buffer);
  Reset(TextF); { Opens the file with a 1K buffer }
  {$I+}
  If (IOresult = 0) then
    Begin
      While NOT(Eof(TextF)) DO
        Begin
          Readln(TextF,Data); Writeln(Data);
        End;
      Close(TextF);
    End
  Else
    Writeln('Can't open file.');
```

End.

RestoreCrtMode procedure **Graph**

Declaration Procedure **RestoreCrtMode**;

Function Clears the screen and turns on the text cursor.

Remark When the text cursor is no longer required, a call to **SetGraphMode** will turn it off.

See also *SetGraphMode*

Example

```
Program RestoreCrtMode_Demo;  
Uses Graph,Crt;  
Var  
    Height,  
    Driver,Mode : Integer;  
    C : Char;  
Begin  
    Driver := DETECT;  
    InitGraph(Driver,Mode,'');  
    RestoreCrtMode; { Turn on text cursor }  
    Write('Enter height of bar 0-',GetMaxY,' : ');  
    ReadLn(Height);  
    SetGraphMode(GetGraphMode); { Turn off text  
cursor }  
    Bar(10,0,30,Height);  
    C := ReadKey;  
    CloseGraph;  
End.
```

ReWrite procedure

Declaration Procedure ReWrite(var F [
FileName:String]);

Function Creates or resets a file.

Remark F is a file variable of any type. **FileName** specifies the name of the file on the disk. The optional **BufSize** can be specified with files of type **text** to determine the size of a buffer.

The ReWrite procedure can be used in two different ways:

ReWrite(MyFile, 'NAME.EXT');

Creates and/or opens the file **MyFile**. If the file is of type **text** it is opened as Write Only. Note that this form is not supported by Turbo Pascal.

ReWrite(MyFile);

Positions the file pointer at the start of **MyFile**. **MyFile** must be open.

See also Assign,Reset, Append,SetTextBuf

Example

```
Program ReWrite_Demo;

Var
  Data,
  Name : String;
  Out : Text;

Begin
  Write('Create which file: ');
  Readln(Name);
  {$I-} Rewrite(Out,Name); {$I+}
  If (IOresult = 0) then
    Begin
      Repeat
        Write('Enter data: ');
        Readln(Data);
        Writeln(Out,Data);
      Until (Data = '');
      Close(Out);
    End
  Else
    Writeln('Could not create ',Name);
End.
```

Rmdir procedure DOS

Declaration Procedure Rmdir(Dir : DirStr);

Function Removes an existing directory.

Remark DirStr is declared in the DOS unit:

Type
DirStr = String[127];

See also ChDir, Mkdir, GetDir

Example Program Rmdir_Demo;

Uses DOS;

```
Begin
  Rmdir(ParamStr(1));
End.
```

Round function

Declaration Function Round(N : Real) : LongInt;

Function Rounds a real-type value to an integer-type value.

See also *Int, Trunc, Abs*

Example Program Round_Demo;

```
Var
  L : LongInt;
Begin
  L := ROUND(1.40); { L = 1 }
  L := ROUND(1.50); { L = 2 }
End.
```

Sector procedure Graph

Declaration Procedure Sector(x,y: Integer;
StAngle,EndAngle,XRadius,YRadius: Word);

Function Draws and fills an elliptical sector.

Remark (X,Y) specifies the centre point of the sector.

StAngle,EndAngle specify the start and end angle and XRadius,YRadius specify the x and y radius of the sector.

The sector is drawn in the current color, and filled using the current fill settings.

See also *SetFillStyle, Arc, Circle, PieSlice, Rectangle, DrawPoly, Ellipse, Bar*

Example Program Sector_Demo;
Uses Graph,Crt;
Var
 Driver,Mode : Integer;
 C : Char;
Begin
 Driver := DETECT;
 InitGraph(Driver,Mode,'');
 Randomize;

```

SetColor(Random(GetMaxColor));
SetFillStyle(SolidFill,Random(GetMaxColor));
Sector(GetMaxX DIV 2,GetMaxY DIV
2,30,330,100,100);
C := ReadKey;
CloseGraph;
End.

```

Seek procedure

Declaration Procedure Seek(var F; Rec : LongInt);

Function Moves the file pointer to a specific record in file F.

Remark F is a file variable (Typed or Untyped) that has been opened.

Rec is the number of the record to seek to. The number of the first record in a file is 0.

See also FilePos, FileSize

Example Program Seek_Demo;

Uses DOS;

```

Var
  F : FILE;
Begin
  Reset(F,ParamStr(1));
  Repeat
    Seek(F,Random(FileSize(F)))
  Until KeyPressed;
  Close(F);
End.

```


SeekEof function

- Declaration** Function SeekEof [(var F : text)] : Boolean;
- Function** As Eof, but skips over all white spaces.
- Remark** F is a file variable of type text, that has been opened. If F is omitted the standard file Input is assumed.
- See also** Eof
- Example** Program SeekEof_Demo;

```
Var
  T : Text;
  X : Integer;
Begin
  Rewrite(T,'SPACES.DTA'); { Make a file with lots
  of spaces. }
  For X := 1 to 1000 DO
    Write(T,#32);
  Reset(T); { Go to start of file. }

  If (SeekEof(T)) then
    Writeln('File contains nothing or only spaces.')
  Else
    Writeln('The file contains data.');
```

```
    Close(T); Erase('SPACES.DTA');
  End.
```

SeekEoln function

- Declaration** Function SeekEoln [(var F : text)] : Boolean;
- Function** As Eoln, but skips over all white spaces.
- Remark** F is a file variable of type text, that has been opened. If F is omitted the standard file Input is assumed.

Eoln

Example

Program Eoln_Demo;

Uses DOS;

Var

F : Text;

C : Char;

Begin

Reset(F,ParamStr(1));

While NOT(Eoln(F)) DO

Begin

Read(F,C);

Write(C);

End;

Close(F);

End.

SetActivePage procedure Graph

Declaration Procedure SetActivePage(Page : Word);

Function Specifies which screen page to work on.

Remark If you have sufficient chip memory you can access as many virtual pages as you wish although they should be used in the order 0,1,2....

Note that Page 0 will always be available since it is the default screen page, created by the InitGraph procedure. On the IBM PC and Atari many modes support 2 pages although some only support 1.

If the page could not be created, due to insufficient memory, any reference to this page will be ignored, and the GraphResult function will return -5 (GrNoLoadMem).

To actually display the page you should use the SetVisualPage procedure.

See also SetVisualPage, GraphResult

Example

```
Program ScreenPage_Demo;
Uses Graph,Crt;
Const
    Mess = 'Press any key to look at page ';
Var
    Driver,Mode : Integer;
    C : Char;
Function RandX : Integer;
Begin
    RandX := Random(GetMaxX);
End;
Function RandY : Integer;
Begin
    RandY := Random(GetMaxY);
End;
Procedure Many_Triangles;
Var
    Poly : Array[1..3] Of PointType;
    Extra : Byte;
Begin
    Repeat
        Extra := Random(100);
        SetFillStyle(Random(UserFill),Random(GetMaxColor+1));
        Poly[1].X := RandX;
        Poly[1].Y := RandY;
        Poly[2].X := Poly[1].X-Extra;
        Poly[2].Y := Poly[1].Y+Extra;
        Poly[3].X := Poly[1].X+Extra;
        Poly[3].Y := Poly[2].Y;
        FillPoly(3,Poly);
        Until KeyPressed;
        C := ReadKey;
    End;
Procedure Many_Bars;
Begin
    Repeat
        SetFillStyle(Random(CloseDotFill)+1,Random(GetMaxColor+1));
        Bar3D(RandX,RandY,RandX,RandY,0,TopOff);
        Until KeyPressed;
        C := ReadKey;
    End;
Procedure Message( Line1, Line2 : String );
Var
    Th,Hx,Hy,W : Integer;
Begin
    Th := TextHeight('M')*2;
    Hx := GetMaxX DIV 2;
    Hy := GetMaxY DIV 2;
```

```

W := (8+TextWidth(Line2) DIV 2);
SetFillStyle(EmptyFill,GetMaxColor);
Bar3D(Hx-W,Hy-Th*1,Hx+W,Hy+Th*2,0,TopOff);
SetTextJustify(CenterText,CenterText);
OutTextXY(Hx,Hy,'PAGE '+Line1);
OutTextXY(Hx,Hy+Th,Line2);
End;
Begin
  Driver := DETECT;
  InitGraph(Driver,Mode,'');
  SetActivePage(1); { Does page 1 exist ? }
  If (GraphResult = GrOK) then
    Begin
      SetActivePage(0);
      Message('0',Mess+'1'); { Display message on page
0. }
      SetActivePage(1); { Start working on page 1. }
      Many_Triangles; { Work, until a key is pressed. }
      Message('1',Mess+'0'); { Write a message. }
      SetVisualPage(1); { Now display page 1. }
      SetActivePage(0); { Start working on page 0. }
      Many_Bars; { Work, until a key is pressed. }
      Message('0','The end'); { Write a message. }
      SetVisualPage(0); { Now display page 0. }
    End
  Else
    OutText('Paging not possible.');
```

Declaration

```

  C := ReadKey;
  CloseGraph;
End.
```

Remark

SetAllPalette procedure Graph

Declaration Procedure SetAllPalette(var Palette);

Function Sets a number of palette entries to a new color.

Remark Palette is a typeless variable, but its structure resembles the structure of the `PaletteType` record:

```

Type
  PaletteType = Record
    Size : Integer;
    Colors : Array[0..15] Of ShortInt;
  End;
```

The first integer specifies the number of colors in `Palette`. The following `N` bytes (`ShortInts`) are the replacement colors to put into the palette.

The number of colors that `Palette` may contain, depends on the current graphics driver and mode:

Name	Palette size
CGAHi, MCGAHi, EGAMonoHi, PC3270Hi, StHigh	2
CGAC0, CGAC1, CGAC2, CGAC3, EGA64Hi, StMedium, AmigaWB	4
EGAHi	8
EGALo, VGAHi, StLow, AmigaWB16	16
AmigaLo32	32

If one of the color values in `Palette` equals -1, the palette entry for that color will not be changed.

See also

SetPalette, SetRGBPalette, GetPalette, GetPaletteSize, GetDefaultPalette

Example

```
Program SetAllPalette_Demo;
Uses Graph,Crt;
Var
  Driver,Mode : Integer;
  C : Char;
  PalData : PaletteType;
Begin
  Driver := DETECT;
  InitGraph(Driver,Mode,'');
  GetPalette(PalData);
  PalData.Colors[0] := Red;
  SetAllPalette(PalData);
  C := ReadKey;
  CloseGraph;
End.
```

SetAspectRatio procedure Graph

Declaration Procedure SetAspectRatio(Xasp, Yasp : Word);

Function Selects the X and Y value used in the aspect ratio calculations.

Remark The aspect ratio is used by Arc, Circle and PieSlice to ensure that the circular figures will appear on the screen as being round and not egg-shaped.

The default values vary according to the mode selected.

See also GetAspectRatio, Arc, PieSlice, Circle

Example

```
Program SetAspectRatio_Demo;
Uses Graph, Crt;
Var
  Driver, Mode : Integer;
  C : Char;
  Xasp, Yasp : Word;
Begin
  Driver := DETECT;
  InitGraph(Driver, Mode, '');
  SetTextJustify(CenterText, CenterText);
  OutTextXY(100, 100, 'Circle');
  Circle(100, 100, 45);
  GetAspectRatio(Xasp, Yasp);
  Dec(Yasp, 5000);
  SetAspectRatio(Xasp, Yasp);
  OutTextXY(200, 100, 'Egg');
  Circle(200, 100, 45);
  C := ReadKey;
  CloseGraph;
End.
```

SetBkColor procedure

Graph

Declaration Procedure SetBkColor(Color : Word);

Function Selects the background color.

Remark The range of the Color parameter depends on the current mode as described under the GetMaxColor function.

See also GetBkColor, SetColor, GetMaxColor, SetPalette

Example

```
Program SetBkColor_Demo;
Uses Graph,Crt;
Var
  Driver,Mode : Integer;
Begin
  Driver := DETECT;
  InitGraph(Driver,Mode,'');
  SetTextJustify(CenterText,CenterText);
  OutTextXY(GetMaxX DIV 2,GetMaxY DIV 2,'Hello');
  Repeat
    SetBkColor(Random(GetMaxColor)); { No reaction on
a mono screen }
  Until KeyPressed;
  CloseGraph;
End.
```

Note that using the HIGH and EXTRA_HIGHLIGHT modes requires keyboard support. If you don't have this, expect some warnings when using the Graph unit calls.

See also InitGraph

SetColor procedure Graph

Declaration Procedure SetColor(Color : Word);

Function Selects the foreground color.

Remark Color specifies what entry in the palette that is to be used as the foreground color. The range of Color depends on the current mode as described under GetMaxColor.

If an illegal value is passed to SetColor, the foreground color will not be changed.

The highest legal color number is returned by a call to the GetMaxColor function.

See also GetColor, GetMaxColor, SetBkColor, SetPalette

Example

```
Program SetColor_Demo;
Uses Graph,Crt;
Var
  Driver,Mode : Integer;
Begin
  Driver := DETECT;
  InitGraph(Driver,Mode,'');
  Repeat
    SetColor(Random(GetMaxColor+1));
    Line(0,0,Random(GetMaxX),Random(GetMaxY));
    Until KeyPressed;
  CloseGraph;
End.
```


SetCustomMode procedure

Graph

Declaration Procedure SetCustomMode(Left,Top,Width,Height, Planes, ViewMode: Integer);

Function Selects the attributes of the AmigaCustom screen mode.

Remark This should be called before InitGraph.

Width and Height specify the pixel width and height of the screen. Left and Top give the offset coordinates of the screen; normally these will be 0.

Planes gives the number of bit planes, thus specify the number of available colors. ViewModes the mode word of the intuition NewScreen structure.

This is a bit orientated variable, the interesting bits here are:

HIRES	\$8000	More than 320 pixels wide
INTERLACE	\$4	
HAM	\$800	Hold and modify
EXTRA_HALFBRITE	\$80	

Note that using the HAM and EXTRA_HALFBRITE modes requires knowledge of the Amiga hardware; if you don't have this, expect some surprises when using the Graph unit calls.

See also InitGraph

SetDate procedure DOS

Declaration Procedure SetDate(Year, Month, Day : Integer);

Function Sets the current date in the operating system.

Remark The legal ranges of Year, Month and Day are:

Year	1980..2099
Month	1..12
Day	1..31

Note that on the Amiga Year may be greater than 2099 or 1978 or 1979, however the ranges above are recommended for compatibility with other systems.

See also *GetDate, SetTime*

Example Program DateTime_Demo;

Uses DOS;

Var

Year, Month, Day, Dow : Word;

Begin

Write('Enter Year : '); Readln(Year);

Write('Enter Month : '); Readln(Month);

Write('Enter Day : '); Readln(Day);

SetDate(Year, Month, Day);

GetDate(Year, Month, Day, Dow);

Writeln('Date has been set to:');

Writeln('Year = ', Year);

Writeln('Month = ', Month);

Writeln('Day = ', Day);

End.

SetDateTime procedure DOS

Declaration Procedure SetDateTime(Var Time: PackedTime);

Function Sets the current time and time via the operating system. This is an Amiga-specific call.

The parameter is a PackedTime value which can be obtained via the PackTime procedure

See also GetTime, SetDate

Example Program SetDateTime_Demo;

Uses DOS;

Var

U:DateTime;

P:PackedTime

Procedure WriteData;

Begin

With U DO

Begin

WriteLn('Time: ',Hour,'.',Min,'.',Sec);

WriteLn('Date: ',Day,'-',Month,'/',Year);

End;

WriteLn;

End;

Begin { main }

With U DO

Begin

Year := 1992;

Month := 12;

Day := 31;

Hour := 23;

Min := 59;

Sec := 59;

End;

WriteLn('Before packing:'); WriteData;

PackTime(U,P);

SetDateTime(P);

End.

SetFAttr procedure DOS

Declaration Procedure SetFAttr(Fil : PathStr; Attr : integer);

Function Sets the attributes of a file.

Remark The file attributes and PathStr are declared in the DOS unit as follows:

```
Const
DeleteFlag = $0001;      {Delete allowed}
ExecuteFlag = $0002;      {Run allowed}
WriteFlag = $0004;        {Write allowed}
ReadFlag = $0008;         {Read allowed}
Archive = $0010;          {Backup flag}
PureFlag = $0020;         {Residentable programs}
ScriptFlag = $0040;       {Set for Script files}
```

```
Type
PathStr = String[127];
```

Note that the following items cannot be set using SetFAttr - they are merely pseudo-attributes that are returned in the SearchRec type.

```
InfoFlag = $0100;        {Set for .info files}
VolumeID = $0200;         {First file in each dir}
Directory = $0400;        {A directory}
AnyFile = $0FFF;          {All flags together}
```

See also GetFAttr

Example

```
Program SetFAttr_Demo;
Uses DOS;
Var
  OldAttr : Integer; { Original file attributes. }

Begin
  Writeln('Working on ',ParamStr(1));
  GetFAttr( ParamStr(1), OldAttr ); { Save original
attributes. }
  SetFAttr( ParamStr(1), OldAttr OR PureFlag ); {
set the pure bit }
  SetFAttr( ParamStr(1), OldAttr ); { Back to
original. }
End.
```

SetFillPattern procedure Graph

Declaration Procedure SetFillPattern(Pattern :
FillPatternType; Color : Word);

Function Specifies the appearance of the user-defined fill pattern.

Remark The FillPatternType array is declared in the Graph unit as:

Type

FillPatternType = Packed Array[1..8] Of
Byte;

Fill patterns are made up of 8*8 bits. When filling is performed, a pixel will be lit on the screen when a bit in the pattern is on.

To fill the screen with small circles, the following fill pattern would be defined:

Binary pattern	Pattern[1..8]
00011100	28 { Pattern[1] }
00100010	34 { Pattern[2] }
01000001	65 { Pattern[3] }
01000001	65 { Pattern[4] }
01000001	65 { Pattern[5] }
00100010	34 { Pattern[6] }
00011100	28 { Pattern[7] }
00000000	0 { Pattern[8] }

Color specifies what fill color to use.

If illegal values are passed to SetFillPattern, GraphResult will return -11 (GrError), and the current fill settings will not be changed.

Otherwise the current fill pattern and fill color will be set equal to the Pattern and Color parameters.

See also

SetFillStyle, GetFillPattern, GetFillSettings, GraphResult, FloodFill, FillEllipse, FillPoly, Bar, Bar3D, PieSlice

Example

```
Program SetFillPattern_Demo;
Uses Graph, Crt;
Var
  Driver, Mode : Integer;
  C : Char;
  NewPattern : FillPatternType;
Begin
  NewPattern[1] := 28;
  NewPattern[2] := 34;
  NewPattern[3] := 65;
  NewPattern[4] := 65;
  NewPattern[5] := 65;
  NewPattern[6] := 34;
  NewPattern[7] := 28;
  NewPattern[8] := 0;
  Driver := DETECT;
  InitGraph(Driver, Mode, '');
  SetColor(GetColor);
  Rectangle(0, 0, GetMaxX, GetMaxY);
  SetFillPattern(NewPattern, GetColor);
  FloodFill(1, 1, GetColor);
  C := ReadKey;
  CloseGraph;
End.
```

SetFillStyle* procedure *Graph

Declaration Procedure SetFillStyle(Pattern, Color : Word);

Function Selects the pattern and the color to use in fill operations.

Remark Pattern specifies the fill pattern to use.

The legal values are declared in the Graph unit as:

```
Const
  EmptyFill      = 0;   {Background color used}
  SolidFill      = 1;   {Solid fill. }
  LineFill       = 2;   { }
  LtSlashFill    = 3;   { \\ thin lines. }
  SlashFill      = 4;   {\\ \\ thick lines. }
  BkSlashFill    = 5;   {/// thick lines. }
  LtBkSlashFill  = 6;   {/// thin lines. }
  HatchFill      = 7;   {Hatch fill. }
  XHatchFill     = 8;   {Cross fill. }
  InterLeaveFill  = 9;   {Very close dots. }
  WideDotFill    = 10;  {Widely dotted fill. }
  CloseDotFill   = 11;  {Close dots. }
  UserFill       = 12;  {User defined fill. }
```

`EmptyFill..CloseDotFill` selects one of the predefined fill patterns.

`UserFill` selects the pattern defined in a call to `SetFillPattern`. If no such call has been made, the user-defined pattern will default to `SolidFill`.

`Color` specifies what fill color to use. The range of `Color` depends on the current graphics driver and mode. If illegal values are passed to `SetFillStyle`, `GraphResult` will return -11 (`GrError`), and the current fill settings will not be changed.

See also *SetFillPattern, GetFillSettings, GetFillPattern, GetMaxColor, FloodFill, FillEllipse, FillPoly, Bar, Bar3D, PieSlice*

Example

```
Program SetFillStyle_Demo;
Uses Graph,Crt;
Var
  Sx,Sy,
  Driver,Mode : Integer;
Begin
  Driver := DETECT;
  InitGraph(Driver,Mode,'');
  Repeat
    SetFillStyle(Random(UserFill),Random(GetMaxColor)+
1);
    Sx := Random(GetMaxX-20);
    Sy := Random(GetMaxY-20);
    Bar(Sx,Sy,Sx+20,Sy+20);
  Until KeyPressed;
  CloseGraph;
End.
```

SetFTime procedure DOS

Declaration Procedure SetFTime(var F; Time : PackedTime);

Function Sets the time and date of a file.

Remark The F parameter is a file variable (Typed, Untyped or Text) that has been assigned a name but is *not* open. This differs from the Turbo Pascal version of the function where, due to the differences in the operating systems, the file must be open.

.The Time parameter is a PackedTime, that has previously been packed using the PackTime procedure or obtained via the GetFTime procedure.

See also GetFTime, PackTime, Reset, ReWrite, Append

Example Program SetFTime_Demo;

```
Uses DOS;

Var
  DiskFile : TEXT;
  DatTim : DateTime;
  PackData : PackedTime;
Begin
  Assign(DiskFile, ParamStr(1));
  Begin
    With DatTim DO
      Begin
        Write('Enter Year : '); ReadLn(Year);
        Write('Enter Month : '); ReadLn(Month);
        Write('Enter Day : '); ReadLn(Day);
        Write('Enter Hour : '); ReadLn(Hour);
        Write('Enter Minute : '); ReadLn(Min);
        Write('Enter Second : '); ReadLn(Sec);
      End;
      PackTime(DatTim, PackData);
      SetFTime(DiskFile, PackData);
      Writeln('Done.');
```

End
End.

SetGraphMode procedure

Graph

Declaration Procedure SetGraphMode(Mode : Integer);

Function Sets the active driver to operate in the specified mode.

Remark Mode specifies the mode to change to. The possible modes with the Amiga driver are:

No	Name
0	CGAC0
1	CGAC1
2	CGAC2
3	CGAC3
4	CGAHi
5	MCGAHi
6	EGALo
7	EGAHi
8	EGA64Hi
9	EGAMonoHi
10	VGAHi
11	PC3270Hi
12	StLow
13	StMedium
14	StHigh
15	AmigaWB
16	AmigaWB16
17	AmigaLo32
18	AmigaCustom

The full details of the modes are given in the section on the Graph unit.

If an invalid mode value is passed to SetGraphMode, GraphResult will return -10 (GrInvalidMode) and the current mode will not be changed.

See also

GetMaxMode, GetGraphMode, RestoreCrtMode, GetDriverName, GetModeName, GetModeRange

Example

```

Program SetGraphMode_Demo;
Uses Graph, Crt;
Var
  Driver, Mode: Integer;
  MaxMode, MinMode: Integer;
  C: Char;
Begin
  Driver:=DETECT;
  InitGraph(Driver, Mode, '');
  GetModeRange(Driver, MinMode, MaxMode);
  For Mode:=MinMode To MaxMode Do Begin
    SetGraphMode(Mode);
    OutText(GetModeName(Mode));
    c:=ReadChar;
  End;
End.

```

SetLineStyle procedure Graph

Declaration Procedure SetLineStyle(LineStyle, Pattern, Thickness : Word);

Function Specifies the pattern and width of lines.

Remark LineStyle specifies which line pattern to use. The legal values are declared in the Graph unit as:

Const		{ Bit-pattern used: }
SolidLn	= 0;	{ 1111111111111111 }
DottedLn	= 1;	{ 1110000011100000 }
CenterLn	= 2;	{ 1111111000111000 }
DashedLn	= 3;	{ 1111111111110000 }
UserBitLn	= 4;	

If LineStyle equals UserBitLn, the 16 bits passed in the Pattern parameter will be used as the line pattern. Otherwise one of the pre-defined line patterns will be used.

Thickness specifies how thick lines are to be. The legal values are declared in the Graph unit as:

```

Const
  NormWidth = 1; { Width is 1 pixel }
  ThickWidth = 3; { Width is 3 pixels }

```

but sadly these are not supported by the Amiga's operating system.

If illegal values are passed to `SetLineStyle`, the `GraphResult` function will return -11 (`GrError`), and the current settings will not be changed.

See also *GetLineSettings, SetWriteMode, Line, LineTo, LineRel*

Example

```

Program SetLineStyle_Demo;
Uses Graph,Crt;
Var
  Driver,Mode,X : Integer;
  C : Char;
Begin
  Driver := DETECT;
  InitGraph(Driver,Mode,'');
  SetWriteMode(XorPut);
  Repeat
    For X := SolidLn to DashedLn Do
      Begin
        SetLineStyle(X,0,NormWidth);
        Line(0,0,GetMaxX,GetMaxY);
      End;
    Until KeyPressed;
    C := ReadKey;
  CloseGraph;
End.

```

SetPalette procedure Graph

Declaration Procedure `SetPalette`(`PalEntry`, `NewColor` : Integer);

Function Sets a specified palette entry to a new color.

Remark `PalEntry` is the palette entry to change. The range of `PalEntry` depends on the current driver and mode, as described under `GetPaletteSize`.

`NewColor` is the new color to put into the palette. The range of `NewColor` is 0..15.

See also
Example

SetAllPalette, SetRGBPalette, GetPaletteSize

```
Program SetPalette_Demo;  
Uses Graph,Crt;  
Var  
  Driver,Mode,X : Integer;  
Begin  
  Driver := DETECT;  
  InitGraph(Driver,Mode,'');  
  Repeat  
    SetColor(Random(GetMaxColor+1));  
    Line(0,0,Random(GetMaxX),Random(GetMaxY));  
  Until KeyPressed;  
  For X := 0 to GetMaxColor Do  
    Begin  
      SetPalette(X,0); { Remove color X from screen }  
      Delay(100);  
    End;  
  CloseGraph;  
End.
```

SetRGBPalette procedure ***Graph***

Declaration Procedure SetRGBPalette(PalEntry, Red, Green, Blue : Integer);

Function Changes the Red-Green-Blue color mix for a specified palette entry.

Remark PalEntry specifies the palette entry to change. The range of PalEntry depends on the current graphics driver and mode as described under GetPaletteSize.

The Red, Green and Blue parameters represent the intensity of the three basic colors, and must all be in the range 0..63. Note that this differs from the standard Amiga range of 0..15; the least significant bits are not supported by the hardware.

If illegal values are passed to SetRGBPalette, no change will be made to the palette.

See also *SetPalette, SetAllPalette, GetPaletteSize*

Example

```
Program SetRGBPalette_Demo;
Uses Graph,Crt;
Var
  RGB,
  Driver,Mode : Integer;
  C : Char;
Begin
  Driver := DETECT;
  InitGraph(Driver,Mode,'');
  SetTextJustify(CenterText,CenterText);
  MoveTo(GetMaxX DIV 2,GetMaxY DIV 2);
  OutTextXY(GetX,GetY-10,'HighSpeed Pascal');
  OutTextXY(GetX,GetY+10,'programming');
  Repeat
    For RGB := 0 to 63 Do
      Begin
        SetRGBPalette(GetColor,RGB,RGB,RGB);
        Delay(5);
      End;
    For RGB := 63 Downto 0 Do
      Begin
        SetRGBPalette(GetColor,RGB,RGB,RGB);
        Delay(5);
      End;
    Until KeyPressed;
    C := ReadKey;
    CloseGraph;
  End.
```

SetTextJustify procedure Graph

Declaration Procedure SetTextJustify(Horiz, Vert : Word);

Function Sets the justification for graphic text output.

Remark Horiz specifies the horizontal justification. The legal values are declared in the Graph unit as:

```
Const
  LeftText    = 0;
  CenterText  = 1;
  RightText   = 2;
```

Vert specifies the vertical justification:

Const

```
BottomText = 0;  
{CenterText = 1;}  
TopText = 2;
```

Text is always centred around a, so called, base-point. When using `OutText` the base-point is the current pointer, when using `OutTextXY` the base-point is the point defined by the specified X and Y co-ordinates.

This is a listing of the various justification settings:

Horizontal justification:

<code>LeftText</code>	Text is output to the left of the base-point.
<code>CenterText</code>	Text is output right on top of the base-point.
<code>RightText</code>	Text is output to the right of the base-point.

Vertical justification:

<code>TopText</code>	Text is output with its top on the base-point.
<code>CenterText</code>	Same as above.
<code>BottomText</code>	Text is output with its bottom on the base-point.

If illegal values are passed to `SetTextJustify`, `GraphResult` will return -11 (`GrError`), and the current justification settings will not be changed.

See also *SetTextStyle, GetTextSettings, OutText, OutTextXY*

Example

```
Program SetTextJustify_Demo;  
Uses Graph,Crt;  
Var  
  ExtraX,  
  Driver,Mode : Integer;  
  C : Char;  
Procedure OutPutX( Data : String );  
Begin  
  OutText(Data); MoveRel(ExtraX,0);  
End;
```

```

Procedure OutPutY( Data : String );
Begin
  OutText(Data);
  MoveTo(GetMaxX DIV 2,GetY+TextHeight('M'));
End;

Begin
  Driver := DETECT;
  InitGraph(Driver,Mode,'');
  SetTextStyle(DefaultFont,HorizDir,2);
  ExtraX := TextWidth('M')*12;
  Line(0,30,GetMaxX,30);
  MoveTo((GetMaxX DIV 2)-ExtraX ,30);
  SetTextJustify(CenterText,TopText );
  OutPutX('TopText');
  SetTextJustify(CenterText,CenterText);
  OutPutX('CenterText');
  SetTextJustify(CenterText,BottomText);
  OutPutX('BottomText');
  Line(GetMaxX DIV 2, 100,GetMaxX DIV 2, 200);
  MoveTo(GetMaxX DIV 2,100);
  SetTextJustify(LeftText ,CenterText);
  OutPutY('LeftText');
  SetTextJustify(CenterText,CenterText);
  OutPutY('CenterText');
  SetTextJustify(RightText ,CenterText);
  OutPutY('RightText');
  C := ReadKey;
  CloseGraph;
End.

```

SetTextStyle procedure Graph

Declaration Procedure SetTextStyle(Font, Direction, CharSize : Word);

Function Selects the style, direction and size of graphic text.

Remark Font specifies the appearance of text. The legal values are declared in the Graph unit as:

```
Const
  DefaultFont      =      0;
  TriplexFont      =      1;      {Uses Extended }
  SmallFont        =      2;      {Uses Italic}
  SansSerifFont    =      3;      {Uses Bold}
  GothicFont       =      4;      {Uses Bold Italic}
```

Direction specifies the direction of text:

```
Const
  HorizDir          =      0; {From left to right }
  VertDir           =      1; {From bottom to top }
```

VertDir is not supported by the Amiga.

CharSize specifies the size of text. The range of CharSize should be 1..7.

If Font is illegal, GraphResult will return -14 (GrInvalidFontNum).

If Direction is illegal -11 (GrError) will be returned.

See also SetTextJustify, GetTextSettings, OutText, OutTextXY

Example

```
Program SetTextStyle_Demo;
Uses Graph,Crt,UtilUnit;
Var
  Xc,Yc,Size : Word;
  Driver,Mode : Integer;
  C : Char;
Begin
  Driver := DETECT;
```



```

InitGraph(Driver,Mode,'');
Xc := 40;
Yc := GetMaxY-40;
MoveTo(Xc,0);
LineTo(GetX,Yc);
LineTo(GetMaxX,GetY);
SetTextJustify(LeftText,BottomText);
SetTextStyle(DefaultFont,VertDir,2);
OutTextXY(Xc,Yc,'Vertical');
SetTextJustify(LeftText,TopText);
SetTextStyle(DefaultFont,HorizDir,2);
OutTextXY(Xc,Yc,'Horizontal');
SetTextJustify(CenterText,TopText);
MoveTo(GetMaxX DIV 2,0);
For Size := 1 to 7 Do
Begin
  SetTextStyle(DefaultFont,HorizDir,Size);
  OutText('Size '+Int2Str(Size));
  MoveRel(0,TextHeight('S'));
End;
C := ReadKey;
CloseGraph;
End.

```

SetTime procedure **DOS**

Declaration Procedure SetTime(Hour, Min, Sec, Sec100 : Word);

Function Sets the current time via the operating system.

Remark The legal ranges of Hour, Min, Sec and Sec100 are:

Hour	0..23
Min	0..59
Sec	0..59
Sec100	0..99

The Sec100 parameter is accurate to one fiftieth of a second on the Amiga and as such the bottom bit is ignored.

See also *GetTime, SetDate*

Example

Program SetTime_Demo;

Uses DOS;

Var

Hour, Minute, Second, Hundred : Word;

Begin

Write('Enter Hours : ');

Readln(Hour);

Write('Enter Minutes : ');

Readln(Minute);

Write('Enter Seconds : ');

Readln(Second);

SetTime(Hour, Minute, Second, Hundred);

GetTime(Hour, Minute, Second, Hundred);

Writeln('Time has been set to:');

Writeln('Hours = ', Hour);

Writeln('Minutes = ', Minute);

Writeln('Seconds = ', Second+Hundred/100:5:2)

End.

SetViewport procedure Graph

Declaration Procedure SetViewport(x1, y1, x2, y2 : Integer; newClip : Boolean);

Function Specifies the boundaries and clipping state of the viewport.

Remark (x1,y1) specifies the upper-left corner, and (x2,y2) the lower-right corner of the viewport. newClip specifies whether or not to clip off any graphic output that exceeds the boundaries of the viewport.

If illegal values are passed to SetViewport, GraphResult will return -11 (GrError), and the current viewport settings will not be changed.

Otherwise GraphResult will return 0 (GrOK), and the current pointer will be placed in the upper-left corner of the new viewport.

See also GetViewSettings, ClearViewport

Example

```
Program SetViewPort_Demo;  
Uses Graph,Crt;  
Var  
  Driver,Mode : Integer;  
  C : Char;  
Begin  
  Driver := DETECT;  
  InitGraph(Driver,Mode,'');  
  Rectangle(0,0,10,10);  
  SetViewPort(20,20,GetMaxX,GetMaxY,ClipOn);  
  Rectangle(0,0,10,10);  
  SetViewPort(200,200,GetMaxX,GetMaxY,ClipOn);  
  Rectangle(0,0,10,10);  
  C := ReadKey;  
  CloseGraph;  
End.
```

SetVisualPage procedure Graph

Declaration Procedure SetVisualPage(Page : Word);

Function Specifies which screen page to display.

Remark Page must be either 0 or 1.

Notes:

Page 0 will always be available since it is the default screen page, created by the operating system.

Page 1 is a virtual screen page, created by the InitGraph procedure.

If InitGraph could not create page 1, due to insufficient space in the heap, any reference to this page will be ignored, and the GraphResult function will return -11 (GrError).

See also SetActivePage, GraphResult

SetWriteMode procedure Graph

Declaration Procedure SetWriteMode(WritMode : Integer);

Function Specifies the way in which lines are to be put on the screen.

Remark The legal values for WritMode are declared in the Graph unit as:

```
Const
    CopyPut      = 0;
    XorPut       = 1;
```

When the write mode is set to CopyPut, it means that lines are simply put on the screen.

When using XorPut, the lines will be XORed (Exclusive ORed) onto the screen, which means that if the line is drawn twice at the same co-ordinates, the original contents of the screen will be restored.

See also DrawPoly, Line, LineTo, LineRel, Rectangle

Example

```
Program SetWriteMode_Demo;
Uses Graph,Crt;
Var
    C,
    Driver,Mode : Integer;
Begin
    Driver := DETECT;
    InitGraph(Driver,Mode,'');
    C := 1;
    SetWriteMode(XorPut);
    Repeat
        Rectangle(C,C,GetMaxX-C,GetMaxY-C); { Draw
rectangle }
        Rectangle(C,C,GetMaxX-C,GetMaxY-C); { Remove it }
    If (C > GetMaxY DIV 2) then
        C := 0
    Else
        Inc(C,2);
    Until KeyPressed;
    CloseGraph;
End.
```

Sin function

Declaration Function Sin(N) : Real;

Function Returns the sine of the parameter.

Remark N is an integer-type or real-type expression. N is assumed to represent an angle in radians.

See also Cos, ValidReal

Example Program Sin_Demo;

```
Var
  R : Real;
Begin
  R := Sin(100);
End.
```

SizeOf function

Declaration Function SizeOf(var Object) : LongInt;

Function Returns the size of an object.

Remark Object can be any variable or type. If the object has been word-aligned, the filler byte(s) are also included in the result that SizeOf returns.

See also FillChar, Move

Example Program SizeOf_Demo;

```
Type
  MyByte=0..255;
Var
  B : Byte;
  MB: MyByte;
  I : Integer;
  L : LongInt;
  R : Real;
  S : String;
```

```

Begin
  Writeln('The size of B is: ',SizeOf(b));
  Writeln('The size of MB is: ',SizeOf(Mb)); {
notice the alignment byte. }
  Writeln('The size of I is: ',SizeOf(i));
  Writeln('The size of L is: ',SizeOf(l));
  Writeln('The size of R is: ',SizeOf(r));
  Writeln('The size of S is: ',SizeOf(s));
End.

```

SPtr function

Declaration Function SPtr : LongInt;

Function Returns the current value of the stack pointer.

See also Ptr

Example Program SPtr_Demo;

```

Begin
  Writeln('Value of stack pointer: ',SPtr);
End.

```

Sqr function

Declaration Function Sqr(N) : (Same type as parameter)

Function Returns the square of N.

Remark N is an integer-type or real-type expression. The result of Sqr(X) is equal to X*X.

See also Sqrt, ValidReal

Example Program Sqr_Demo;

```

Var
  R : Real;
Begin
  R := Sqr(10);
End.

```

Sqrt function

Declaration Function Sqrt(N) : Real;

Function Returns the square root of N.

Remark N is an integer-type or real-type expression.

See also Sqr, ValidReal

Example Program Sqrt_Demo;

```
Var
  R : Real;
Begin
  R := Sqrt(4);
End.
```

Str procedure

Declaration Procedure Str(x [:width[:decimals]]; var
Res : String);

Function Converts a numeric value to a string.

Remark By specifying Width and Decimals it is possible to make Res look like the output generated by Write and WriteLn.

See also Val, Copy

Example Program Str_Demo;

```
Var
  R : Real;
  S : String;
Begin
  R := 1234.5678;
  WriteLn(R :10:4);
  Str(R :10:4, S);
  WriteLn(S);
End.
```

Succ function

Declaration Function Succ(X) : (Same type as parameter)

Function Returns the successor of the parameter.

Remark X is an ordinal-type value.

See also Pred, Inc, Dec

Example Program Succ_Demo;

Begin

WriteLn(Succ(1)); { = 2 }

End.

Swap function

Declaration Function Swap(N : Integer) : Integer;

Function Swaps the high and low bytes of N.

See also SwapWord, Hi, Lo

Example Program Swap_Demo;

Var

X : Integer;

Begin

X := Swap(\$1234); { = \$3412 }

End.

SwapWord function

Declaration Function SwapWord(N : LongInt) : LongInt;

Function Swaps the high and low words of N.

See also Swap, HiWord, LoWord

Example Program SwapWord_Demo;

```
Var
  X : LongInt;
Begin
  X := SwapWord($12345678);
  { = $56781234 }
End.
```

TextBackground procedure Crt

Declaration Procedure TextBackground(color:Integer);

Function Changes the background color of Crt text.

Remark The color parameter should be between 0 and 3. Note that the standard Workbench colors are used rather than those on an IBM PC

See also TextColor

Example program TextBackground_Test;
Uses CRT;
var i:integer;
begin
 TextBackground(3);
 Write('This is on colour 3');
 TextBackground(0);
 Writeln('This is back on colour 0');
end.

TextColor procedure Crt

Declaration Procedure TextColor(color:Integer);

Function Changes the foreground color of Crt text.

Remark The color parameter should be between 0 and 3. Note that the standard Workbench colors are used rather than those on an IBM PC

See also TextBackground

Example

```
program TextColor_Test;
Uses CRT;
var i:integer;
begin
  for i:=1 to 3 do begin
    TextColor(i);
    WriteLn('Colour ',i);
  end;
  TextColor(1);
end.
```

TextHeight procedure Graph

Declaration Function TextHeight(TextString : string)
: Word;

Function Returns the height, in pixels, of a text string.

Remark The value returned by TextHeight depends on the current font settings.

TextHeight, together with the TextWidth function, is useful when centring text on the screen.

See also TextWidth, OutText, OutTextXY, Graph

Example

```
Program TextDimension_Demo;
Uses Graph,Crt;
Var
  Driver,Mode : Integer;
  C : Char;
```

```

Begin
  Driver := DETECT;
  InitGraph(Driver,Mode,'');
  C := 'A';
  Repeat
    OutText(C);
    MoveRel(TextWidth(C),TextHeight(C));
  C := Succ(C);
  Until (GetY >= GetMaxY);
  C := ReadKey;
  CloseGraph;
End.

```

TextWidth procedure Graph

Declaration Function TextWidth(TextString : string) : Word;

Function Returns the width, in pixels, of a text string.

Remark The value returned by TextWidth depends on the current font settings.

TextWidth, together with the TextHeight function, is useful when centring text on the screen.

See also TextHeight, OutText, OutTextXY, Graph

Trunc function

Declaration Function Trunc(N : Real) : LongInt;

Function Truncates a real-type value to an integer-type value.

Remark Trunc returns the value of N rounded down towards 0.

See also Int, Round, Abs

Example

```

Program Trunc_Demo;
Var
  L : LongInt;
Begin
  L := TRUNC(1234.5678);
End.

```

UnPackTime procedure

Declaration Procedure UnPackTime(Time : PackedTime;
var Result : DateTime);

Function Unpacks a PackedTime item into a human readable DateTime record.

Remark Packs a 16-byte DateTime record into a PackedTime structure.

Remark The DateTime record is declared in the DOS unit:

```
DateTime = RECORD
    DayOfWeek      : Word;
    Year           : Word;
    Month          : Word;
    Day            : Word;
    Hour           : Word;
    Min            : Word;
    Sec            : Word;
    Sec100         : Word;
END;
```

PackedTime corresponds to the type tDateStamp in the AmigaDOS unit and is used in calls to SetFTime, GetFTime and UnPackTime procedures.

Note that on the Atari and under MS-DOS a LongInt is used rather than PackedTime.

See also PackTime

UpCase function

Declaration Function UpCase(Ch : Char) : Char;

Function Converts the letter Ch to upper-case.

Remark UpCase will only return a result different from Ch, if Ch is in the range of 'a' .. 'z'.

See also ReadKey

Example

Program UpCase_Demo;

```
Var
  C : Char;
Begin
  Repeat
    Write('Enter a letter. ');
    C := ReadKey;
    WriteLn(' You pressed ',UpCase(C));
  Until (C = #13);
End.
```

Val procedure

Declaration Procedure Val(S : String; var R; var ErrorPos : Integer);

Function Converts a string to a numeric variable.

Remark R is an integer-type or real-type variable. Upon return ErrorPos holds the position of the first character in S that could not be converted.

See also Str, Copy

Example Program Val_Demo;

```
Var
  R : Real;
  S : String;
  Error : Integer;
Begin
  Write('Enter a number: '); ReadLn(S);
  Val(S,R,Error);
  If (Error = 0) then
    WriteLn('OK')
  Else
    WriteLn('Error in number at position ',Error,
  !');
End.
```

ValidReal function

Declaration Function ValidReal(R) : Boolean;

Function Returns true if R is a valid real value.

Remark Use this function to test the results of calculations done with real-type values.

Example

```
Program ValidReal_Demo;
Var
  R : Real;
Begin
  R := Ln(0); { Cannot be done. }
  If (ValidReal(R)) then
    Writeln('OK')
  Else
    Writeln('Error in calculation.');
```

End.

WhereX function Crt

Declaration Function WhereX:Integer;

Function Returns the X co-ordinate of the cursor in the current Crt unit window.

Remark If the cursor is at the left hand edge of the window 1 is returned.

See also WhereY,GotoXY

WhereY function Crt

Declaration Function WhereY:Integer;

Function Returns the Y co-ordinate of the cursor in the current Crt unit window.

Remark If the cursor is at the top of the window 1 is returned.

See also WhereY,GotoXY

WindMax function Crt

Declaration Function WindMax:WORD;

Function Returns the bottom right hand corner of the current Crt unit window.

Remark Lo(WindMax) gives the X character co-ordinate and Hi(WindMax) gives the Y character co-ordinate. Note that, unlike the other cursor positioning commands, this is based on (0,0) as the top left rather than (1,1). Thus to position the cursor at the bottom left you would use:

```
GotoXY(Lo(WindMax)+1,Hi(WindMax)+1);
```

Note that unlike the Turbo Pascal version this is a function rather than a variable.

See also WindMin,GotoXY

Example

```
Program WindMax_Demo;  
Uses Crt;  
Begin  
  WriteLn(' bottom right is',  
    Lo(WindMax),',',Hi(WindMax));  
End.
```

WindMin function Crt

Declaration Function WindMax:WORD;

Function Returns the top right hand corner of the current Crt unit window.

Remark Lo(WindMin) gives the X character co-ordinate and Hi(WindMin) gives the Y character co-ordinate. Note that, unlike the other cursor positioning commands, this is based on (0,0) as the top left rather than (1,1). In fact this always returns 0 on the Amiga but is provided for those who wish to write programs that are portable to/from the PC.

See also WindMax,GotoXY

Write procedure

Declaration Procedure Write [([var F;] v1
[:Width:Deci] [,v2,..vn])];

Function Writes 1 or more components to file F.

Remark Write for text files:

F is a file variable of type **text**. If omitted the standard file **Output** is assumed. Each of the components to write must be of type **char**, **integer**, **real**, **string** or **boolean**.

Each of the components can be formatted using the two format parameters **Width** and **Deci**. **Width** specifies the minimum length of the components. **Deci** specifies the number of decimals the components are to have.

Write for typed files:

F is a file variable of any type except **text**. Each of the components (v1,v2,vn) must be of the same type as the component type of file F.

See also WriteLn,Read,ReadLn

Example Program Write_Demo;

```
Var
  R : Real;
  X : Byte;
Begin
  R := 1234.5678;
  For X := 10 to 30 DO
    Begin
      Write(R :X:X); Writeln;
    End;
  End.
```


WriteLn procedure

Declaration Procedure WriteLn [(var F : text; v1
[:Width:Deci] [,v2,..vn])];

Function Executes the Write procedure, and then writes an end-of-line marker to the file.

Remark Writeln can only be used on file variables of type text.

See also Write,ReadLn,Read

Example Program WriteLn_Demo;
Var

R : Real;

X : Byte;

Begin

R := 1234.5678;

For X := 10 to 30 DO

WriteLn(R :X:X);

End.

The File Attribute Constants

WriteFlag := 0001;	(Delete allowed)
ExecuteFlag := 0002;	(Run allowed)
WriteFlag := 0004;	(Write allowed)
ReadFlag := 0008;	(Read allowed)
Volume := 0010;	(Backup flag)
ShareFlag := 0020;	(Shareable program)
ScriptFlag := 0040;	(Set for Script files)
InfoFlag := 0100;	(Set for .info files)
VolumeID := 0200;	(First file in each dir)
Directory := 0400;	(A directory)
AnyFile := 08FFF;	(All flags together)
DirFlags := 'dewrapsIVD';	(Flags as shown by List)

The InfoFlag, VolumeID and Directory flags are pseudo-flags with the following meanings:

Info_flag indicates that the filename ends in .info.

Chapter 3

The Supplied Units

The DOS Unit

The DOS unit implements a variety of operating system and disk related units. It corresponds directly to the DOS unit of Turbo Pascal and the Atari version of HighSpeed Pascal. However because of the greater differences between the Amiga operating system and MS-DOS there are inevitably some incompatibilities which we have tried to minimise whilst still letting you access the facilities of AmigaDOS without calling it directly.

The precise details of the procedures and functions provided by this unit are given in alphabetical order in the *Procedure Reference* chapter, but this section gives an overview of the data structures and facilities of the DOS unit.

The File Attribute Constants

DeleteFlag	= \$0001;	{Delete allowed}
ExecuteFlag	= \$0002;	{Run allowed}
WriteFlag	= \$0004;	{Write allowed}
ReadFlag	= \$0008;	{Read allowed}
Archive	= \$0010;	{Backup flag}
PureFlag	= \$0020;	{Residentable programs}
ScriptFlag	= \$0040;	{Set for Script files}
InfoFlag	= \$0100;	{Set for .info files}
VolumeID	= \$0200;	{First file in each dir}
Directory	= \$0400;	{A directory}
AnyFile	= \$0FFF;	{All flags together}
DirFlags	= 'dewraps?IVD';	{Flags as shown by List}

The InfoFlag, VolumeID and Directory flags are pseudo-flags with the following meanings:

Info_flag indicates that the filename ends in .info.

VolumeID is set for the first directory in a search, thus giving the name of the volume.

Directory indicates that an item is a directory.

These attributes are used when manipulating the attributes of files via the SetFAttr and GetFAttr routines. They are also used in the directory manipulation routines.

```
ReadOnly   = $1000; {Unsupported}
Hidden     = $2000; {Unsupported}
Sysfile    = $4000; {Unsupported}
```

The above constants correspond to the appropriate flags under MS-DOS and on the Atari; they are not supported under AmigaDOS.

In general it is best to use AnyFile as this is portable across systems.

File Handling String Types

```
Type
ComStr      = String[127];
PathStr     = String[127];
DirStr      = String[127];
NameStr     = String[ 31];
ExtStr      = String[ 15];
C_Str       = Packed Array[0..255] OF Char; {C style}
```

These string types are used as parameters for the procedures that take file and path names. ChDir, RmDir, Mkdir and FExpand are examples of such routines. Note that these types are larger than the equivalents used under MS-DOS and on the Atari as Amiga filenames are more flexible and often considerably larger. When porting programs that use arrays of these types from other systems be aware that considerably more storage is required on the Amiga.

The SearchRec type

SearchRec = RECORD

Lock	: Longint;	{*Internal use only}
Attr	: Integer;	{Flag of file found}
Size	: LongInt;	{Size of found file}
Time	: PackedTime;	{Date&Time}
Name	: NameStr;	File found}
Reserved	: tFileInfoBlock;	{* }
ResAttr	: Integer;	{* Attr wanted}
ResPattern	: NameStr;	{* Pattern wanted}
END;		

The SearchRec type is used when searching for files using the FindFirst and FindNext routines. Those items marked * above are for internal use only and are subject to change.

The Date and Time types

DateTime = RECORD

DayOfWeek	: Word;
Year	: Word;
Month	: Word;
Day	: Word;
Hour	: Word;
Min	: Word;
Sec	: Word;
Sec100	: Word;
END;	

PackedTime =tDateStamp;

This type is used to provide a 'human-readable' version of the packed time representations that are used by the operating system and the GetFtime, SetDateTime and SetFtime routines. To convert between items of type PackedTime and DateTime you can use the PackTime and UnpackTime procedures.

Under MS-DOS and on the Atari, packed dates and times are stored in items of type LongInt. So you will need to change the types of such variables when porting programs across operating systems.

DosError variable

The `DosError` variable will contain a non-zero AmigaDOS error number if an error occurs during a DOS unit operation. The meanings of these error numbers are given in *Appendix A*.

PatternProc variable

The `PatternProc` variable is used to control whether AmigaDOS style pattern matching (via `ADosPattern`) or MSDOS style pattern matching (via `MsDosPattern`) is used by the `FindFirst` and `FindNext` procedures.

Process-Handling Procedure

<code>Exec</code>	runs another program with a given command line.
-------------------	---

Environment functions

<code>EnvCount</code>	returns the number of environment strings (i.e. files) in <code>ENV:</code> .
-----------------------	---

<code>GetEnv</code>	returns the value (contents) of a given environment variable.
---------------------	---

<code>EnvStr</code>	returns the <i>n</i> th environment string - useful if you wish to display the values of the current environment.
---------------------	---

Command line handling functions

<code>ParamCount</code>	the number of parameters in the command line of the current program.
-------------------------	--

<code>ParamStr</code>	returns the <i>N</i> th parameter in the command line.
-----------------------	--

Directory handling procedures

ChDir	changes the current directory.
MkDir	creates a new directory.
RmDir	removes an existing directory.
GetDir	finds the current directory.
FSplit	splits a path into the directory name, file name and file extension.
FExpand	takes a file name and expands it into a full path and file name string.
FExpandLock	takes an AmigaDOS Lock and expands it into a full path and file name. This is an Amiga specific function.

Date and Time procedures

GetDate	returns the current date according to the operating system.
GetTime	returns the current time according to the operating system.
SetDate	sets the current time according to the operating system.
SetTime	sets the current time according to the operating system.
SetDateTime	sets the date and time simultaneously. Note that this is an Amiga specific call.
GetFtime	reads the date/time stamp of a particular file.
SetFtime	sets the date/time stamp of a particular file.

PackTime	packs a DateTime record into a PackedTime record.
UnpackTime	expands a PackedTime record into a DateTime record.

File Handling procedures and functions

FindFirst	searches a directory to find the first file name that matches and returned the corresponding SearchRec.
FindNext	finds the next file name matching the specification given in a call to FindFirst.
StopFindFirst	a routine that must be called when not reading all the files that are given by FindFirst/FindNext. This is an Amiga specific routine.
SetFattr	sets the attributes (see the <i>File Attribute</i> constants above).
GetFattr	returns the attributes of a file.
ADosPattern	is used to implement AmigaDOS-style pattern matching for FindFirst/FindNext, when its address is assigned to PatternProc.
MSdosPattern	is used to implement MSDOS-style pattern matching for FindFirst/FindNext, when its address is assigned to PatternProc.

Miscellaneous function

LockAlertWindow	this function can be used to suppress the appearance of alerts asking for disk insertions when searching for files.
-----------------	---

The System unit

The system unit provides some of the facilities that are built-in to ISO Pascal compilers together with many of system specific facilities of HighSpeed Pascal. All HighSpeed Pascal programs automatically 'use' the system unit so the procedures and variables can be used as if they were built into the compiler.

The System types

Type Byte=0..255;

The **Byte** type is a sub-range specifying the legal range of variables of type **Byte**. Note that you could not actually define **Byte** yourself since otherwise it would use 2 bytes of memory rather than 1.

Type Word=0..65535;

The **Word** type is a sub-range specifying the legal range of variables of type **Word**.

The file record types

Type

Char128 = Packed Array[0..127] of Char;
PChar128 = ^Char128;

Type

TextRec = Record

fInpFlag:	Boolean;	{Used for input}
fOutFlag:	Boolean;	{Used for output}
fHandle:	LongInt;	{File handle}
fBufSize:	Integer;	{Rbuffer size}
fBufPos:	Integer;	{buffer position}
fBufEnd:	Integer;	{buffer last used}
fBufPtr:	PChar128;	{buffer pointer}
fInOutProc:	Pointer;	{IO handler if fHandle is zero}
fUser:	Longint;	{Unused }
fName:	Array [1..64] of Char;	{File name (ASCIIIZ)}
fBuffer:	Char128;	{the default buffer }
end;		

The `TextRec` type is used for files of type `Text` and is discussed in more detail in the Appendix *Inside HighSpeed Pascal*.

Type

```
FileRec = Record
  fInpFlag: Boolean; {Used for input}
  fOutFlag: Boolean; {Used for output}
  fHandle: LongInt; {File handle}
  fBufSize: Integer; {Record size}
  fPrivate: Array[1..6] of Integer;
  fUser: Longint; {Unused}
  fName: Array [1..64] of Char; {File name (ASCIIIZ)}
end;
```

The `FileRec` type is used as a file control block for typed and untyped files.

The Input and Output variables

```
Var Input, Output : Text;
```

The `Input` and `Output` variables are the standard `Input` and `Output` devices, used for input from the keyboard and output to the screen. They are also the default files for the text handling routines.

The Stack variables

```
Var HighStak, LowStack: Pointer;
```

These variables give the top and bottom of the stack respectively.

The HeapOrg variable

```
Var HeapOrg: Pointer;
```

This points to the very bottom of the heap.

The DevChain variable

```
Var DevChain: Pointer;
```

This is used by the system module to maintain the list of user defined devices that have been added using the `Device` procedure.

The RandSeed Variable

Var RandSeed: LongInt;

This is used as the seed for the next random number generated by the Random function.

The ExitProc variable

Var ExitProc: Pointer;

The ExitProc variable points to the chain of routines that are executed at the end of a program.

The ErrorAdd Variable

Var ErrorAdd: Pointer;

This variable contains the offset from the start of the program of the instruction that caused a program to give a runtime error. This can be read when an ExitProc procedure is being executed.

The ExitCode variable

Var ExitCode: Integer;

This contains the error number (either AmigaDOS or Pascal) when a runtime error occurs.

The IOResVar variable

Var IOResVar: Integer;

IOResVar is the variable that is read and cleared by the IOResult function.

The LastPC variable

Var LastPC: LongInt

This variable is updated by the stack and range checking routines to give the PC value of the most recent piece of code that has been executed.

The SysBase variable

`Var SysBase: Pointer;`

SysBase contains the Amiga Exec library base pointer.

The DosBase variable

`Var DosBase: Pointer;`

DosBase contains the Amiga DOS library base pointer.

The RunFromMemory variable

`Var RunFromMemory: Boolean;`

This variable is true if the program has been run from memory within the integrated environment rather than from disk.

The Crt unit

The Crt unit provides the facilities of an old style terminal but in a window on the Amiga screen. You can position the cursor at a particular place before using the conventional `write` and `writeln` commands, clear the rest of the window etc. It also provides keyboard input a character at a time rather than the line at a time that is used by the system unit `read` and `readln` commands.

We have made this unit as compatible as possible with the Turbo Pascal PC implementation of the Crt unit whilst using the default Amiga Shell/CLI window. This means that your program cannot re-size the window nor are multiple windows supported. On the other hand the user can re-size the window and, if you are using `read/readln` for input, type a line ahead just like any other Amiga CLI program. Under Workbench 2 you can also use the cursor and history features that are provided by the Shell. Also note that the colours used by `TextBackground` and `TextColor` are the current Amiga ones rather than as they would be on a PC.

Screen output procedures

ClrScr	clears the CLI window and places the cursor in the top left corner.
ClrEos	clears the CLI window starting at the current cursor position.
ClrEol	clears to the end of the current line.
InsLine	inserts a line; causing the lines below to scroll down.
DelLine	deletes a line causing the lines below to scroll up.
GotoXY	moves the cursor to a particular screen position.

Text effect procedures

TextBackground	sets the background text colour of the current CLI window.
TextColor	sets the text foreground colour.
HighVideo	uses bold; the nearest equivalent to high intensity video.
NormVideo	returns to normal output after using HighVideo.

Keyboard Input functions

KeyPressed	returns true if the user has pressed a key ready for reading via ReadKey.
ReadKey	reads a character from the keyboard without echo; if KeyPressed is true this will be read immediately otherwise it waits for a key.

Status Inquiry functions

WhereX	returns the current cursor X co-ordinate.
WhereY	returns the current cursor Y co-ordinate.
WindMin	(Lo(WindMin),Hi(WindMin)) are the co-ordinates of the top left of the window.
WindMax	(Lo(WindMin),Hi(WindMin)) are the co-ordinates of the bottom right of the window.

Miscellaneous procedure

AssignCrt	makes i/o to a file go to the Crt rather than to a file.
-----------	--

The Graph unit

The Graph unit enables you to write programs using equivalent facilities to those provide by the Borland Graphics Interface for the PC. These routines have the advantage of portability and ease of programming but do not provide access to the entire range of facilities provided by the Amiga. To do this you will need to call the Amiga's operating system directly.

Graph unit error codes

```
GrOK = 0; {No error }
GrNoInitGraph = -1; {Graph has not been initialized }
GrNotDetected = -2; {No graphics adapter detected }
GrFileNotFound = -3; {}
GrInvalidDriver = -4; {Bad driver for h/w configuration}
GrNoLoadMem = -5; {}
GrNoScanMem = -6; {}
GrNoFloodMem = -7; {}
GrFontNotFound = -8; {}
GrNoFontMem = -9; {}
GrInvalidMode = -10; {Bad mode for graphics driver }
GrError = -11; {Generic error }
GrIOError = -12; {}
GrInvalidFont = -13; {}
GrInvalidFontNum = -14; {Invalid font number requested }
GrNoGraphLib = -15; {Could not open graphics library }
GrNoIntuiLib = -16; {Could not open intuition }
GrNoLayersLib = -17; {could not open layers }
GrAreaFail = -18; {error in AreaFill operation }
```

The constants above that are followed by {} are not used on the Amiga but are provided for compatibility with the PC. The last three items are Amiga specific.

Graphics driver constants

```
DETECT = 0; { Returns standard Amiga}
CGA = 1;
MCGA = 2;
EGA = 3;
EGA64 = 4;
EGAMono = 5;
IBM8514 = 6;
HercMono = 7;
ATT400 = 8;
VGA = 9;
SVGA = 10;
PC3270 = 11;
StColor = 12;
StMono = 13;
Amiga = 14; { Standard Amiga RGB-monitor }
```

The only driver number that is used on the Amiga is the last one; the others are supplied for compatibility with the PC and Atari.

Resolution numbers

CGAC0	=	0;
CGAC1	=	1;
CGAC2	=	2;
CGAC3	=	3;
CGAHi	=	4;
MCGAC0	=	0;
MCGAC1	=	1;
MCGAC2	=	2;
MCGAC3	=	3;
MCGAMed	=	4;
MCGAHi	=	5;
EGALo	=	6;
EGAHi	=	7;
EGA64Lo	=	6;
EGA64Hi	=	8;
EGAMonoHi	=	9;
HercMonoHi	=	7;
ATT400C0	=	0;
ATT400C1	=	1;
ATT400C2	=	2;
ATT400C3	=	3;
ATT400Med	=	4;
ATT400Hi	=	5;
VGALo	=	6;
VGAMed	=	7;
VGAHi	=	10;
PC3270Hi	=	11;
IBM8514Lo	=	10;
IBM8514Hi	=	10;
StLow	=	12;
StMedium	=	13;
StHigh	=	14;
AmigaWB	=	15;
AmigaWB16	=	16;
AmigaLo32	=	17;
AmigaCustom	=	18;

The majority of these names are only supplied for compatibility with other machines.

The modes that are supported on the Amiga together with their approved names are as follows:

No	Name	Width	Height	Planes
0	CGAC0	320	200	2
1	CGAC1	320	200	2
2	CGAC2	320	200	2
3	CGAC3	320	200	2
4	CGAHi	640	200	1
5	MCGAHi	640	480	1
6	EGALo	640	200	4
7	EGAHi	640	350	3
8	EGA64Hi	640	350	2
9	EGAMonoHi	640	350	1
10	VGAHi	640	480	4
11	PC3270Hi	640	256	1
12	StLow	320	200	4
13	StMedium	640	400	2
14	StHigh	640	400	1
15	AmigaWB	640	256	2
16	AmigaWB16	640	256	4
17	AmigaLo32	320	256	5
18	AmigaCustom	640	256	2

The figures in the above table for the Amiga resolutions are the defaults on a PAL Amiga; the actual values used are 'clones' of the Workbench screen (with AmigaLo32 using half the width). Thus on an NTSC machine the height will normally be 200 pixels. If over-scan is being used then these values will be larger. The values for the Amiga custom mode given above are the defaults if SetCustomMode has not been called.

The MCGAHi, EGALo, EGAHi, EGA64Hi, EGAMonoHi, VGAHi, StMedium and StHigh modes are all interlaced.

Note that the PC3270Hi, VGAHi and MCGAHi modes have the disadvantage that because the lower portion of the screen is not visible on NTSC machines.

The first four modes differ in the palettes that they use, viz:

CGAC0	Light Green, LightRed, Yellow
CGAC1	LightCyan, LightMagenta, White
CGAC2	Green, Red, Brown
CGAC3	Cyan, Magenta, LightGray

Colour numbers

Black	=	0;
Blue	=	1;
Green	=	2;
Cyan	=	3;
Red	=	4;
Magenta	=	5;
Brown	=	6;
LightGrey	=	7;
DarkGrey	=	8;
LightBlue	=	9;
LightGreen	=	10;
LightCyan	=	11;
LightRed	=	12;
LightMagenta	=	13;
Yellow	=	14;
White	=	15;

These are the default colours for most modes.

EGA colour constants

EGABlack	=	0;
EGABlue	=	1;
EGAGreen	=	2;
EGACyan	=	3;
EGARed	=	4;
EGAMagenta	=	5;
EGABrown	=	20;
EGALightGrey	=	7;
EGADarkGrey	=	56;
EGALightBlue	=	57;
EGALightGreen	=	58;
EGALightCyan	=	59;
EGALightRed	=	60;
EGALightMagenta	=	61;
EGAYellow	=	62;
EGAWhite	=	63;

These constants are for compatibility only; they are used to provide EGA emulation on the IBM 8514 graphics adapter.

Line style Constants

These line constants are used by `GetLineSettings` and `SetLineStyle` to specify the width of lines.

<code>SolidLn</code>	<code>=</code>	<code>0;</code>
<code>DottedLn</code>	<code>=</code>	<code>1;</code>
<code>CenterLn</code>	<code>=</code>	<code>2;</code>
<code>DashedLn</code>	<code>=</code>	<code>3;</code>
<code>UserBitLn</code>	<code>=</code>	<code>4;</code>
<code>NormWidth</code>	<code>=</code>	<code>1;</code>
<code>ThickWidth</code>	<code>=</code>	<code>3;</code>

Font Style Constant

<code>DefaultFont</code>	<code>=</code>	<code>0;</code>
<code>TriplexFont</code>	<code>=</code>	<code>1;</code>
<code>SmallFont</code>	<code>=</code>	<code>2;</code>
<code>SansSerifFont</code>	<code>=</code>	<code>3;</code>
<code>GothicFont</code>	<code>=</code>	<code>4;</code>
<code>HorizDir</code>	<code>=</code>	<code>0;</code>
<code>VertDir</code>	<code>=</code>	<code>1;</code>
<code>UserCharSize</code>	<code>=</code>	<code>0;</code>

These font styles are used by `GetTextSettings` and `SetTextStyle`.

Justification constants

<code>LeftText</code>	<code>=</code>	<code>0;</code>
<code>CenterText</code>	<code>=</code>	<code>1;</code>
<code>RightText</code>	<code>=</code>	<code>2;</code>
<code>BottomText</code>	<code>=</code>	<code>0;</code>
<code>TopText</code>	<code>=</code>	<code>2;</code>

These horizontal and vertical justification used by `SetTextJustify`.

The Clipping constants

ClipOn = True;
ClipOff = False;

The clipping constants are used to specify the clipping status of a view port via SetViewport.

Miscellaneous constants

TopOn = True;
TopOff = False;

These are used in calls to Bar3D to control whether a bar is to have a top.

Fill pattern constants

EmptyFill = 0; { Background color is used. }
SolidFill = 1; { Solid fill. }
LineFill = 2; { -- }
LtSlashFill = 3; { \\\ thin lines. }
SlashFill = 4; { \\\ thick lines. }
BkSlashFill = 5; { /// thick lines. }
LtBkSlashFill = 6; { /// thin lines. }
HatchFill = 7; { Hatch fill. }
XHatchFill = 8; { Cross fill. }
InterLeaveFill = 9; { Very close dots. }
WideDotFill = 10; { Widely dotted fill. }
CloseDotFill = 11; { Close dots. }
UserFill = 12; { User defined fill. }

These fill patterns are used by GetFillSettings and SetFillStyle.

Put mode constants

```
NormalPut = 0; { Replace }
CopyPut   = 0; { Replace }
XorPut     = 1; { XOR }
```

The above constants are used by PutImage and SetWriteMode.

```
OrPut      = 2; { OR }
AndPut     = 3; { AND }
NotPut     = 4; { NOT }
```

The above constants are used by PutImage only.

The MaxColors constant

```
MaxColors = 15;
```

Used to define the PaletteType record.

PaletteType record

```
PaletteType= Record
    Size      : Integer;
    Colors    : Array[0..MaxColors] Of ShortInt;
End;
```

Used by GetPalette, GetDefaultPalette and SetAllPalette.

LineSettingsType record

```
LineSettingsType= Record
    LineStyle  : Word;
    Pattern    : Word;
    ThickNess  : Word;
End;
```

The LineSettingsType is used by GetLineSettings to return the current line settings.

TextSettingsType record

```
TextSettingsType= Record
    Font      : Word;
    Direction : Word;
    CharSize  : Word;
    Horiz     : Word;
    Vert      : Word;
End;
```

The TextSettings type is used by GetTextSettings to return the current text settings.

FillSettingsType record

```
FillSettingsType= Record
    Pattern    : Word;
    Color      : Word;
End;
```

The FillSettingsType record is used by GetFillSettings to return the current fill settings.

FillPattern Type record

```
FillPatternType = Packed Array[1..8] Of Byte;
```

The FillPatternType type is used in calls to SetFillPattern and GetFillPattern. It contains a user-defined fill pattern.

PointType record

```
PointType = Record
    X,Y    : Integer;
End;
```

The PointType record is used in calls to DrawPoly and FillPoly.

ViewPortType record

```
ViewPortType = Record
    X1,Y1,X2,Y2 : Integer;
    Clip        : Boolean;
End;
```

The ViewPortType type is used in calls to GetViewSettings.

ArcCoordsType record

```
ArcCoordsType = Record  
  X,Y      : Integer;  
  XStart   : Integer;  
  YStart   : Integer;  
  XEnd     : Integer;  
  YEnd     : Integer;  
End;
```

This type is used by **GetArcCoords**.

Aspect Routines

GetAspectRatio	returns the aspect ratio settings.
SetAspectRatio	sets the aspect ratio.

Bar routines

Bar	draws and fills a bar.
Bar3D	draws and fills a 3-dimensional bar.

Color and Palette Routines

GetBkColor	returns the current background color.
GetColor	returns the current foreground color.
GetDefaultPalette	returns the Graph unit default palette.
GetMaxColor	returns the maximum color number for the current graphics mode.
GetPalette	returns the current palette and its size.
GetPaletteSize	returns the size of the current palette.
SetAllPalette	sets a number of palette entries to a new color.
SetBkColor	sets the background color.
SetColor	sets the foreground color.

SetPalette	sets a specified palette entry for a new color.
SetRGBPalette	sets the RGB color mix for a specified palette entry.

Device routines

ClearDevice	clears the entire screen.
CloseGraph	shuts down the Graph unit.
DetectGraph	checks the hardware to find a legal graphics driver and mode.
GetDriverName	returns the name of the current graphics driver.
GetGraphMode	returns the current graphics mode.
GetMaxMode	returns the maximum mode for the current graphics driver.
GetMaxX	returns the highest legal X co-ordinate.
GetMaxy	returns the highest legal Y co-ordinate.
GetModeName	returns the name of the current graphics mode.
GetModeRange	returns the range of modes for the current graphics driver.
GraphDefaults	resets the Graph unit to its default settings.
InitGraph	starts up the Graph unit.
RestoreCrtMode	clears the screen and turns on the text cursor.
SetActivePage	selects the screen page to display.
SetGraphMode	sets the operational mode of the current graphics driver.
SetVisualPage	selects the screen page to display.

Error Handling routines

GraphResult	returns the error code for the last executed Graph unit command.
GraphErrorMsg	returns an explanation to a specified Graph unit error code.

Fill routines

FillEllipse	draws and fills an ellipse.
FloodFill	fills an area surrounded by a specified color.
GetFillPattern	returns the current user-defined fill pattern.
GetFillSettings	returns the current fill settings.
SetFillPattern	specifies the appearance of the user-defined fill pattern.
SetFillStyle	selects the pattern and color to use in fill operations.

Image routines

GetImage	saves a specified area of the screen.
ImageSize	returns the size of the specified screen area.
PutImage	puts a saved screen area onto the screen.
FreeImage	frees the memory that has been allocated via GetMem and then used by GetImage .

Line routines

GetLineSettings	returns the current line settings.
Line	draws a line.

LineRel	draws a line relative to the current position.
LineTo	draws a line from the current position to the specified position.
SetLineStyle	specifies the pattern and width of lines.

Pixel routines

GetPixel	returns the color of a specified pixel.
PutPixel	sets a specified pixel to a specified color.

Poly routines

DrawPoly	draws a polygon.
FillPoly	draws and fills a polygon.
GetTextSettings	returns the current text settings.
OutText	outputs a string of text at the current position.
OutTextXY	outputs a string of text at the specified position.
SetTexJustify	sets the justification for graphic text output.
SetTextStyle	sets the style, direction and size of graphic test.
TextHeight	returns the height of a string of text.
TextWidth	returns the width of a string of text.
ClearViewport	clears the current viewport.
GetViewSettings	returns the current viewport settings.
SetViewport	specifies the viewport settings.

Appendix A

AmigaDOS Error Codes

This Appendix details the numeric AmigaDOS errors and their meanings. You may also use the AmigaDOS **FAULT** command followed by an error number to obtain the meaning.

103 insufficient free store
out of memory.

105 task table full
limit of 20 CLIs.

114 bad template

115 bad number

116 required argument missing

117 keyword requires argument

118 too many arguments

120 argument line invalid or too long
when using CLI commands.

121 file is not an object module
trying to execute non-executable file.

122 invalid resident library during load

201 no default directory

- 202 object in use
such as a file by another program.
- 203 object already exists
- 204 directory not found
- 205 object not found
most commonly a file.
- 206 invalid window
in name specification.
- 207 object too large
- 208 invalid action
- 209 packet request type unknown
- 210 invalid stream component name
name too long or contains control chars.
- 211 invalid object lock
- 212 object not of required type
such as directory name instead of file
- 213 disk not validated
disk is still being validated, or bad.
- 214 disk write-protected
- 215 rename across devices attempted
- 216 directory not empty
when trying to delete it.

217 too many levels

218 device not mounted

after specifying a volume name.

219 seek error

219 seek error

220 comment too big

file comments must be less than 80.

221 disk full

222 file is protected from deletion

223 file is protected from writing

224 file is protected from reading

225 not a DOS disk

226 no disk in drive

232 no more entries in directory

233 object is soft linked

234 object is linked

235 bad hunk

236 not implemented

240 record not locked

241 lock collision

242 lock timeout

243 unlock failed

303 Buffer overflow
in internal or user buffer.

304 break received

305 file not executable
E bit is cleared.

Appendix B

Error Messages

Compiler Errors

The first class of error messages are those where the compiler is expecting a particular symbol but it cannot find it. These are as follows:

';' expected

':' expected

',' expected

('' expected

'),' expected

['' expected

]'' expected

.' expected

'=' expected

':'= expected

...' expected

END expected

DO expected

OF expected

THEN expected

TO or DOWNT0 expected

Note that simply inserting the appropriate character sometimes won't be sufficient as this sort of error can be caused by an error earlier in the program.

BEGIN expected

The compiler expects BEGIN when it cannot find any further declarations in a function. This is probably caused by a mis-formed declaration.

INTERFACE expected

The reserved word UNIT must follow a Unit heading.

IMPLEMENTATION expected

This is usually given with the cursor on garbage text or the word BEGIN. The compiler expects IMPLEMENTATION when it cannot find any further declarations.

The following are also 'expected' errors , but are more complex than those described above.

Constant expected

Boolean expression expected

The construct IF *expression* THEN requires a boolean expression. You cannot write IF 2+3 THEN, for example.

Integer constant expected

The real assignment R:=123.4; is allowed, but the assignment R:=12345678; is not because the compiler first reads the number as an integer. You can add a .0 to make it a real.

Integer expression expected

Integer variable expected

Integer or real constant expected

Integer or real expression expected

Integer or real variable expected

Pointer variable expected

Typed Pointer variable expected

File variable expected

Record variable expected

Ordinal type expected

Ordinal expression expected

String constant expected

String expression expected

String variable expected

Identifier expected

Variable expected

Type identifier expected

Field identifier expected

Char expression expected

The remainder of the compiler's error messages are more varied:

Unknown identifier

Undefined type in pointer definition.

'.' expected after module name

When using unit references the unit name must be followed by a period (.).

Duplicate identifier

Note that the program name is also entered into the symbol table. This can be used to reference items in other units if redefined by other units. If you have made a Reset procedure, the you can only reset a file by writing `System.Reset(aFile)`.

Type mismatch

Constant out of range

Constant and CASE types do not match

Operand types do not match operator

Invalid result type

A function can only return simple, string and real types.

Invalid string length

Invalid subrange base type

Lower bound greater than upper bound

Invalid FOR control variable

The FOR variable must either be declared in same function or global. It must not be part of any structure.

Illegal assignment

String constant exceeds line

You may have forgotten an apostrophe in the string. Note that a single apostrophe in a string must be written as two apostrophes.

Error in integer constant

An integer constant that is out of range has been declared.

Error in real constant

An integer constant that is out of range has been declared.

Division by zero

The compiler has attempted to divide something by zero.

Structure too large

Records (but not arrays) may not exceed 32K bytes in size.

Constants are not allowed here

Invalid type cast argument

Type casting can only be performed when the argument is of the same size as the desired type.

Invalid '@' argument

Only works on functions, procedures and variables.

Invalid GOTO

A label has been declared (LABEL MyLabel;) but does not actually occur in the main block (MyLabel:).

Label not within current block

It is not possible to jump to a label outside the current block.

Undefined label

The target of a GOTO has not been defined.

Label already defined

An attempt to re-define a label has been made.

Invalid file type

Cannot read or write variables of this type

Files must be VAR parameters

They may not be passed by value.

File components may not be files

Set base type out of range

A set base type must be a subrange with its bounds in the range 0..255 or an enumerated type with no more than 256 possible values.

Error in type

Error in statement

Error in expression

Undefined FORWARD procedure: xx

Undefined EXTERNAL procedure: xx

Invalid external definition, symbol = xx

Invalid external reference, symbol = xx

Too many symbols

Only 32 Kbyte is set aside for the symbol table for each unit or program.

Too many nested scopes

Also given when too many units are used. Maximum is 63.

Expression too complicated

All data registers in the 68000 processor are used for temporary integers.

Too many variables

There is a limit of 32K of symbol table per module and you have exceeded this. Split the code into units.

Too much code

No more than 32K of code is allowed per module. Split the code into units.

Bad unit or program name: xx

Unit missing: xx

Unit not found

The linker cannot find a specific unit in the PASCAL.LIB file or using the directories specified via the appropriate compiler option.

Incompatible unit versions

The program is trying to use a unit compiled with another version of the compiler than that you are currently using. If you receive an upgrade from us you will need to remove any existing .UNI files.

Syntax error

Something is wrong!

Unexpected end of text

You may have more BEGINS than ENDS. Maybe caused by an non-terminated comment.

Line too long

Only 127 characters are allowed on one line.

Invalid compiler directive

An unknown compiler directive has been found.

Bad object file in file: xx

You cannot make BSR and BRA to external defined labels. You *must* use JSR and JMP. See the restrictions in *Inside HighSpeed Pascal*.

Not enough memory

You can increase the size of its workspace using the -Knn or compiler option or the Workspace item from the Compiler Settings requester. If you then run out of memory when the compiler is initialising up you can reduce the size of the resident library (pascal.lib) and/or remove other programs that you have running and files from your ramdisk.

Too many files (\$I or USES)

Each Uses file uses an include level when it is re-compiled.

No room in 16 bit fix-up field, try with \$F+

If a unit has to call a procedure or function in another unit which is more than 32K bytes away then the compiler has to use a 32 bit offset. The \$F+ tells the compiler to make the calls using this format.

Duplicate unit name: xx

A unit name has been specified twice in the same Uses clause. e.g. Uses Dos,Dos.

Conditional variable missing

The name of a conditional variable is missing in a \$DEFINE, \$UNDEF, \$IFDEF or \$IFNDEF directive.

Misplaced conditional directive

For example using a \$ELSE directive when there is no current \$IFDEF directive.

ENDIF directive missing

Either this error or the Unexpected End of text error will occur if an \$ENDIF directive is missing after a \$IFDEF or \$ELSE directive.

Runtime Errors

AmigaDOS error numbers are returned 'as is' and correspond to the messages given in *Appendix A*. These are the errors generated by the HighSpeed Pascal runtime system itself.

Runtime errors are generated when your program is actually executing and cause the program to terminate. There are as follows:

50 Stack overflow.

Generated by the \$S+ control code. The check is performed at entry to every procedure and function when enabled.

Fix: Your program has got too little stack space, or it has entered a recursive procedure that did not terminate before the stack area was used up. Your procedures may also have large structures (arrays or records) as local variables, which are located on the stack while the procedure is running.

51 Range check.

Generated by the additional \$R+ code.

Fix: You may be trying to assign an invalid value to a variable, either by using an assignment as := or passing it to a procedure/ function. Another possible bug is trying to access a nonexistent element of an array or string.

52 Bus error.

Generated by the 680x0 CPU, when trying to access invalid memory.

Fix: The problem could be that you (by accident of course) are using a pointer variable containing garbage.

53 Address error.

Generated by the 68000 CPU, when trying to access word or longword elements from an odd address. This cannot be done in one instruction, you have to read it one byte at a time. Note that this is not generated on the 68020 and higher processors.

Fix: You will only see this problem if you make the addresses yourself, by making your own pointer variables with the `Ptr` function. HighSpeed Pascal always locates its word variables on even addresses.

54 Other errors.

Bus, address and `DIV0` errors are hardware traps in the 68000 system. There are a lot of other traps defined. If the program terminates due to a trap other than these three, this error is shown.

55 `DIV0`. Division by zero.

Whenever the 68000 CPU makes a division by zero, this error is given. Long integers are divided by internal routines, which also give error 55 when you try divide by zero.

56 Heap overflow.

When the heap is so full that a request for a new block from it cannot be satisfied, this error is given.

Fix: You must test the `MaxAvail` function instead of the `MemAvail` function. `MaxAvail` returns the size of the largest block where `MemAvail` returns the total.

The following are errors are given by the file system:

60 File not open.

You may not perform input/output on a file that has not been opened via `reset` or `rewrite`.

61 File not open for read.

You can only write to a text file in `rewrite` mode, but not read from it.

62 File not open for write.

You can only read from a text file in reset mode, but not write to it. Perhaps you should use append first.

63 No memory for device.

A HighSpeed Pascal device driver needs heap memory for its buffer and name.

64 Read Write error.

A read/write error has occurred.

65 Bad numeric value read.

For example, 1.23e3e3e is a bad value.

Memory Layout

The memory layout of a loaded user program looks like that normally used on the Amiga. There are three separate blocks:

- **Code**, all your code and the runtime code
- **Data**, used for structured constants
- **BSS**, used for the global area

In addition, when a program starts it allocates two further areas of memory:

- **Stack**, used for storing local variables, temporary values and for procedure linkage. The size of the stack can be set using the first parameter to the `SM` compiler directive.
- **Heap**, the area of memory that is used for calls to the Pascal `NEW` and `GETMEM` procedures. The size of the heap will be between the `hmin` and `hmax` parameters to the `SM` directive, with at least free (the last parameter to `SM`) bytes left for the operating system and other programs.

The following variables are set up by the initialization code:

heaporg Pointer to the bottom of the heap

highstack Top of stack

lowstack Bottom of stack. The `stack check` routine checks the stack pointer against this.

53 File not open for write, error 53

When you attempt to open a file for writing, the system may return this error if the file does not exist or if you do not have write permission for the file. To correct this error, make sure the file exists and that you have the necessary permissions.

54 Read/write error. A read/write error has occurred. This error may occur if the disk is full, if the disk is damaged, or if the disk is not properly formatted. To correct this error, check the disk status and format it if necessary.

55 Other errors.

This error message is used for various other errors that may occur. The specific error code is given in the message. To correct the error, refer to the appropriate error message.

56 DIV0: Division by zero.

When you attempt to divide a number by zero, the system returns this error. This is an illegal operation. To correct this error, make sure the divisor is not zero.

58 Heap overflow.

When the heap is full, but a request for a new block is given, the system returns this error. This indicates that the program has exceeded the available heap space.

Fix: You must limit the size of the heap. Use the `HEAPSIZE` parameter in the `LINK` command to specify the maximum size of the heap.

The following errors are given by the file system:

60 File not open.

You may not perform input/output on a file that has not been opened. To correct this error, open the file first.

61 File not open for read.

You can only write to a file that is open in write mode. To correct this error, open the file in read mode.

53 File not open for write.

You have tried to write to a file that has not been opened for writing. You must use the `FILE_WRITE` option when opening the file.

Fix: Use the `FILE_WRITE` option when opening the file.

54 Head Write Error. A head write error has occurred.

55 Other errors.

56 Bad numeric value read. A bad numeric value has been read from the file.

57 File not open for read. You have tried to read from a file that has not been opened for reading.

58 DIV0. Division by zero.

You have tried to divide by zero. This is an illegal operation.

59 Heap overflow.

You have tried to allocate more memory than is available. This is an illegal operation.

60 File not open.

You have tried to open a file that does not exist.

61 File not open for read.

You have tried to read from a file that has not been opened for reading.

62 File not open for write.

You have tried to write to a file that has not been opened for writing.

63 File not open.

You have tried to open a file that does not exist.

64 File not open for read.

You have tried to read from a file that has not been opened for reading.

65 File not open for write.

You have tried to write to a file that has not been opened for writing.

66 File not open.

You have tried to open a file that does not exist.

Appendix C

Inside HighSpeed Pascal

Here is the information needed for hackers and those who need a bit more specific information on the internal workings of HighSpeed Pascal. All variables mentioned in this chapter are located in the System unit.

Memory Layout

The memory layout of a loaded user program looks like that normally used on the Amiga. There are three separate hunks :

- Code, all your code and the runtime code
- Data, used for structured constants
- BSS, used for the global area

In addition when a program starts it allocates two further areas of memory:

- Stack, used for storing local variables, temporary values and for procedure linkage. The size of the stack can be set using the first parameter to the **\$M** compiler directive.
- Heap, the area of memory that is used for calls to the Pascal **NEW** and **GetMem** procedures. The size of the heap will be between the **hmin** and **hmax** parameters to the **\$M** directive with at least **free** (the last parameter to **\$M**) bytes left for the operating system and other programs.

The following variables are set up by the initialisation code:

HeapOrg Pointer to the bottom of the heap.
HighStak Top of Stack
LowStack Bottom of stack. The stack check routine checks the stack pointer against this.

The `LowStack` is 126 bytes above the real bottom. It can be incremented a bit if you need to have a larger margin because one of your procedures uses a lot of local variables.

Traps and Exceptions

When a program starts, it initializes some traps so that an error message can be given if an exceptions occurs and `ErrorHandler` procedures are called.

Exit Procedures

The system unit implements a variable `ExitProc: Pointer`. It points to a procedure that is called when a program is about to terminate, no matter what the cause is.

The `ExitProc` is initiated to point to the system's exit handler, which removes the installed traps before the final termination. If this handler is not called before the program terminates, you are asking for trouble. The installed traps will point to non existent code after the program has terminated if they are not removed.

The `ExitProc` pointer enables you to gain control before the program terminates, just by entering your own exit procedure into the chain. Your exit procedure can then close files and do other cleaning jobs. When the `ExitProc` handler starts working, it keeps calling the procedure pointer to by `ExitProc`, until the last procedure (the system's own exit) has been called.

You use the Exit handler system this way:

```
var
  OldExitProc: Pointer;
Procedure MyExitHandler;
begin
  ExitProc:=OldExitProc;
  Cleanup...
end;

procedure Initialize;
begin
  OldExitProc:=ExitProc;
  ExitProc:=@MyExitHandler;
end;
```

The exit handler should restore the `ExitProc` variable as the first thing. Any errors that occur before the `ExitProc` is restored will terminate the program right away, skipping the execution of the systems exit handler.

When the exit handler starts it work, it resets the stack pointer to `HighStak`. Otherwise a stack overflow might occur.

Heap Manager

The heap is where dynamic data is kept. Data blocks are allocated with `New` and `GetMem`, and disposed of again using `Dispose` and `FreeMem`. Every block released must be released by using the same parameters for the release as for the allocation. If more memory is released than retrieved the heap system will be destroyed.

Heap memory is allocated in blocks of 8 byte at a time. Allocating 1000 integers on the heap will use 1000×8 bytes of the heap, the same space as for 1000 double reals.

When memory blocks are released in the middle of other blocks, there will be a hole, that can later be reused. If there are no holes in the heap, the result of `MemAvail` will equal the result of `MaxAvail`. The free list is kept in the first 8 bytes of a released block. Initially there is one big free hole.

Data formats

Integer Types

Values are stored using as much space as needed to save all the information including a sign bit.

Integers, $-32768..32767$, are stored in a 16 bit word.

LongInt's, $-2147483648..2147483647$, are saved in a 32 bit (long) word.

ShortInt's, $-128..127$, are saved in an 8 bit word.

Subranges of integers are saved using signed values. Subranges that lay between the bounds of ShortInt, Integer or LongInt are stored using one of these formats. A subrange of 0..255 therefore uses a 16 bit word.

The types Byte and Word are exceptions to this rule. Although the range of a Byte is 0..255 they are stored in a single 8-bit byte.

Packed structures store a subrange of 0..255 as an unsigned 8 bit byte.

Char Types

The predefined type Char, #0..#255 are stored using a 16 bit word, with a zero in the high byte, and the ASCII value of the character in the low byte.

A subrange in the area from #0..#127 is stored in an 8 bit byte.

Packed structures do save a Char as a an unsigned 8 bit byte.

Enumerated Types

An enumerated type is stored as a byte if it contains less than 129 values, else it is saved using a 16 bit word.

Boolean Types

The Boolean type is an enumerated type of False..True and as such saved in a byte.

Real Types

The Single and Double types are saved in 4 and 8 bytes using the IEEE standard format as it can be found in a 68881 manual. An extended real is saved in 10 bytes using an internal format. This format is not compatible with the 68881 Floating Point Unit. Note that at the time of writing all real calculations are performed by internal routines rather than via Amiga system libraries as the latter do not support greater than 8 byte precision.

This is what you get from the reals:

Type	Bytes	Range	Useful digits
Single	4	3.4e-38..3.4e38	7-8
Real=Double	8	1.7e-308..1.7e308	15-16
Extended	10	1.1e-4932..1.1e4932	18-19

The trigonometric functions all work using **Double** precision. Do not use **Reals** for counting. In integer types, the LOWER xx bits are saved, always giving you access to add one in order to change the value. The real types only saves the UPPER xx bits of the value, skipping the lesser significant bits.

Calculations on real types never cause runtime errors. Instead the result returned contains a special mask. By using the function **ValidReal**, you can test if a returned value is valid as this function will return **false** if its parameter is illegal.

```
Function ValidReal(R: Real): Boolean;
```

The predefined constant **NAN** contains the special mask used for signaling bad reals. When writing a value containing a **NAN**, the string **NAN** or **-NAN** is written.

Pointer Types

A Pointer is stored using a 32 bit longword. **NIL** is represented as a longword of zero. The **Ptr** function takes a **LongInt** and type converts it into a pointer. The 32 bit value returned is the same 32 bits as in the 32 bit **LongInt**.

Set Types

Set types are stored using 1 to 32 bytes. Each byte can hold eight elements. The lower and upper limits of the set are byte aligned before calculating the number of bytes needed.

A set of 6..10 uses 2 bytes because the limits are aligned to the range of 0..15.

The bit number within a byte is calculated as: $\text{Number} = \text{ElementNumber} \bmod 8$.

Array Types

Unpacked elements are all stored on even addresses, forcing byte elements to be interleaved with filler bytes. A packed array packs Char and Integer subranges of 0..255 into unsigned byte elements.

The array elements are stored one after the other in memory (maybe with padding) with the element with the lowest indices first in memory. A multi dimensional array is saved with the right most dimension increasing first.

Example:

Var A3: array[1..3,1..10,1..20] of integer.

The array A3[1,3] is an array of [1..20] of integer.

The array A3[2] is an array of [1..10,1..20] of integer.

Record Types

The first element of the record is stored first in memory followed by the others in same sequence as written. All variables using more than one byte are word aligned. All variant parts start at the same address.

As with arrays, packed records does pack Char and Integer subranges of 0..255 into unsigned byte elements.

File Types

Text files use 216 bytes of memory. This is the memory layout of the text file control block:

Type

```
TextRec = Record
  fInpFlag: Boolean; {Used for input}
  fOutFlag: Boolean; {Used for output}
  fHandle: LongInt; {File handle}
  fBufSize: Integer; {Rbuffer size}
  fBufPos: Integer; {buffer position}
  fBufEnd: Integer; {buffer last used}
  fBufPtr: PChar128; {buffer pointer}
  fInOutProc: Pointer; {IO handler if fHandle is zero}
  fUser: Longint; {Unused }
  fName: Array [1..64] of Char;{File name (ASCIIIZ)}
  fBuffer: Char128; {the default buffer }
end;
```

fBufPtr points to the text file's buffer. By default this is contained with the record itself (at fBuffer). You can allocate your own buffer using the SetTextBuffer call.

Typed and untyped files use 88 bytes using the following structure:

Type

```
FileRec = Record
  fInpFlag: Boolean; {Used for input}
  fOutFlag: Boolean; {Used for output}
  fHandle: LongInt; {File handle}
  fBufSize: Integer; {Record size }
  fPrivate: Array[1..6] of Integer;
  fUser: Longint; {Unused }
  fName: Array [1..64] of Char;{File name (ASCIIIZ)}
end;
```

Note that the field that are common to both text and other files are in exactly the same positions, so you can use pointers to either type should you need to read them directly.

Calling Conventions

Parameters for procedure (and function) calls are moved to the stack using the normal Pascal order of parameters. The first parameter meet in the source is first moved onto the stack. This is opposite to the C-language way of doing things. The called routine removes all parameters from the stack, except for the result of a functions.

A procedure call looks like this:

```
MOVE Param1, -(SP)
MOVE Param2, -(SP)
MOVE Param3, -(SP)
JSR  Procedure
```

A function call looks like this:

```
SUB   #nn,SP
MOVE Param1, -(SP)
MOVE Param2, -(SP)
MOVE Param3, -(SP)
JSR   Function
MOVE (SP)+,Result
```

The function call first makes room for the result on the stack, and removes the result itself after the call. If the result uses more than 4 bytes, then only the address is passed. This is the sequence used:

```
PEA   Result
MOVE Param1, -(SP)
MOVE Param2, -(SP)
MOVE Param3, -(SP)
JSR   Function
ADD   #4,SP
```

The procedure and function code looks like this:

```
JSR   StackCheck

LINK A6,#-StackUsed
MOVEM.L D3-D7/A2-A5, -(SP)
{Take copy of value parameters}
...
MOVEM.L (SP)+,D3-D7/A2-A5
UNLK A6
MOVE.L (SP)+,A0
ADD   #ParamSize,SP
JMP   (A0)
```

The last 3 lines may be replaced by a single `rts` if the routine takes no parameters. The `movem` is removed if none of the registers `d3-d7` or `a2-a5` are used by the routine. The `StackCheck` routine is called if the compiler directive `$$+` is active. It looks at the `StackUsed` parameter, and compares this value added with the stack pointer register, `SP`., against the system variable `StackLow`.

If the called procedure is nested, that is if it is declared inside another procedure, the stack frame is also passed as if it were the final parameter of the procedure. All parameters then move four bytes up in memory from the A6 register. A procedure call to a nested procedure looks like this:

```
MOVE Param1, -(SP)
MOVE.L A6, -(SP)
JSR Procedure
```

The procedure can then use code as:

```
MOVE.L 4(A6), A0
MOVE nn, (A0)
```

in order to read or write variables in other levels. If the nesting level is more than one, a number of `MOVE.L 4(A0), A0` can be inserted between these two lines.

Value Parameters

Value parameters get copied every time the routine gets called. If the size of the variable is less or equal to 4 bytes, its value is pushed onto the stack straightaway.

When the size is greater than 4 bytes a reference is pushed. A normal Pascal routine then makes its own copy as the first thing in the routine. An assembler routine does not have to take a copy if it does not change the parameter, thereby saving time and code.

Two and four byte parameters are passed using the same storage method as used when storing in normal variables. Byte values are passed using the normal procedures for bytes on the stack, using a word. Otherwise the stack pointer would be misaligned.

Real types are passed by reference to an extended real, maybe first converted from the other real formats.

Arrays and Records are passed by reference unless they can be contained in four bytes. Strings are passed by reference. Sets are passed as references to a maximized set of 256 elements.

Variable Parameters

Variable parameters are always passed by pushing the address of the variable. The routine then uses the exact same location for its formal parameter as the caller does for its actual variable. Both variables must, of course, be of exact same type.

Function Results

For function results of Integer, Char, Boolean and any subrange, two or four bytes are allocated on the stack before the call. The result is returned in this space and removed after the call.

For Real types, the caller allocates 10 bytes for an extended real, and pushes the address of this before any parameters. The caller removes this again after the call.

For String types, the caller allocates temporary space for the string, pushes the address of this before any parameters and removes it again after the call. Other types cannot be returned.

Using Assembly Language

Inline assembler

HighSpeed Pascal enables you to type assembly source directly into your Pascal programs. The full 68000 instruction set is supported and you can use Pascal constants and variables but not the instructions that were added with the 68020,030 etc. Using the inline assembler makes it easy to write small sections of inline code but does not give you the wealth of directives and power of macros provided by a full assembler like HiSoft Devpac.

The inline assembler is accessed via the `ASM` statement whose syntax is as follows:

```
ASM [Label:] AsmStmnt <Separator AsmStmnt> END;
```

where `Label` is an optional label and `AsmStmt` is one assembler statement. The `Separator` is either a semi-colon or a new-line. Notice that this is different from the rest of the HighSpeed Pascal compiler where a new line is only equivalent to a space. You may have more than one `AsmStmt` on a single line separated by semi-colons or on separate lines as is customary with assembly language.

For example:

```
Function Add3(X,Y,Z: Integer):Integer;  
begin  
  ASM  
    move.w    X,d0  
    add.w     Y,d0  
    add.w     Z,d0  
  END.
```

Registers

The assembler supports the standard 68000 register names:

Data registers: D0,D1,D2,D3,D4,D5,D6,D7

Address registers: A0,A1,A2,A3,A4,A5,A6,A7,SP

Special registers: USP,PC,CCR,SR

Labels, Constants and variables

Labels declared in a Pascal `Label` definition can be used and/or defined in an ASM statement. However the inline assembler also supports its own local labels. These are prefixed with an @ sign and do not need to be declared before use.

There is also a special label `@result`. This represents the position of the result of the current function. This will either be the place to store the result or a pointer to the result, as described above under *Calling Conventions*.

You can use Pascal constants and variables from with your assembler statements. For local variables and parameters the assembler will automatically supply the appropriate addressing mode; you should not code this explicitly.

The assembler does not 'understand' about the sizes of variables. Thus

ASM

```
move MyLongInt,d0
```

END

gives only the upper word of MyLongInt in the lower half of the D0 register.

Operators

The following operators are supported by the assembler:

Highest precedence:

+	Unary plus
-	Unary minus. Returns the value negated
!	Unary negation. Returns the bitwise not value.

Second highest precedence

*	Multiplication
/	Integer division
%	Integer remainder
&	Bitwise AND
<<	Logical shift left
>>	Logical shift right

Lowest precedence:

+	Addition
-	Subtraction
	Bitwise OR
^	Bitwise XOR

Addressing modes:

The following are the 68000 CPU addressing modes as supported by the inline assembler:

Form	Meaning	Example
Dn	data register direct	D3
An	address register direct	A5
(An)	address register indirect	(A1)
(An)+	address register indirect with post-increment	(A5)+
-(An)	address register indirect with pre-decrement	-(A0)
d(An)	address register indirect with displacement	20(A7)
d(An,Rn.s)	address register indirect with index	4(A6,D4.L)
d	absolute short address	\$0410
d	absolute long address	\$12000
d(PC)	program counter relative with offset	@NEXT(PC)
d(PC,Rn.s)	program counter relative with index	NEXT(PC,A2.W)
#d	immediate data	#26

In the above table:

n denotes register number from 0 to 7

d denotes a number

R denotes index register, either A or D

s denotes size, either W or L, when omitted defaults to W

When using address register indirect with index the displacement may *not* be omitted, for example, instead of

```
move.l (a3,d2.l),d0
```

use

```
move.l 0(a3,d2.l),d0
```


Addressing modes:

The following are the 68000 CPU addressing modes as supported by the inline assembler:

Form	Meaning	Example
Dn	data register direct	D3
An	address register direct	A5
(An)	address register indirect	(A1)
(An)+	address register indirect with post-increment	(A5)+
-(An)	address register indirect with pre-decrement	-(A0)
d(An)	address register indirect with displacement	20(A7)
d(An,Rn.s)	address register indirect with index	4(A6,D4.L)
d	absolute short address	\$0410
d	absolute long address	\$12000
d(PC)	program counter relative with offset	@NEXT(PC)
d(PC,Rn.s)	program counter relative with index	NEXT(PC,A2.W)
#d	immediate data	#26

In the above table:

- n denotes register number from 0 to 7
- d denotes a number
- R denotes index register, either A or D
- s denotes size, either W or L, when omitted defaults to W

When using address register indirect with index the displacement may *not* be omitted, for example, instead of

```
move.l (a3,d2.l),d0
```

use

```
move.l 0(a3,d2.l),d0
```

Instructions

The following are the 68000 assembly language instructions as supported by the inline assembler:

ABCD	ADD	ADDA	ADDI	ADDQ
ADDX	AND	ANDI	ASL	ASR
BCC	BCHG	BCLR	BCS	BEQ
BGE	BGT	BHI	BHS	BLE
BLO	BLS	BLT	BMI	BNE
BNZ	BPL	BRA	BSET	BSR
BTST	BVC	BVS	BZE	CHK
CLR	CMP	CMPA	CMPI	CMPM
DBCC	DBCS	DBEQ	DBF	DBGE
DBGT	DBHI	DBHS	DBLE	DBLO
DBLS	DBLT	DBMI	DBNE	DBNZ
DBPL	DBRA	DBT	DBVC	DBVS
DBZE	DIVS	DIVU	EOR	EORI
EXG	EXT	ILLEGAL	JMP	JSR
LEA	LINK	LSL	LSR	MOVE
MOVEA	MOVEM	MOVEP	MOVEQ	MULS
MULU	NBCD	NEG	NEGX	NOP
NOT	OR	ORI	PEA	RESET
ROL	ROR	ROXL	ROXR	RTE
RTR	RTS	SBCD	SCC	SCS
SEQ	SF	SGE	SGT	SHI
SHS	SLE	SLO	SLS	SLT
SMI	SNE	SNZ	SPL	ST
STOP	SUB	SUBA	SUBI	SUBQ
SUBX	SVC	SVS	SWAP	SZE
TAS	TRAP	TRAPV	TST	UNLK

Note that the short branches (BRA.S etc) are not supported. The only way to use an address in the code is via LEA/PEA or by jumping or calling it. The assembler will automatically convert this to the appropriate PC relative addressing modes. The assembler does not support relocation in the code section.

DC Directive

The only directive supported is DC for constant bytes. There are three possible sizes .B, .W and .L. DC.B blocks always generate an even number of bytes.

Examples:

ASM

```
DC.B 11, 'Hello World'      {Pascal String}
DC.B 'Hello World', 0
{C string }
DC.B 1, 2, 3+4+5
DC.W 7, 9, 13
DC.L 1
```

END;

Another example

It is important to remember that assembly language expressions are evaluated at compile time rather than run-time. Suppose we want to code the following in assembly language:

Const X=4;

 Y=7;

Var

 M, N, 0: Integer;

 A: Array [0..99] of Integer;

.....

M:=X+Y;

M:=N+X;

M:=A[3];

The following is the correct way:

ASM

```
MOVE.W        #X+Y, M        {M:=X+Y}
MOVE.W        N, D0    {M:=N+X}
ADD.W        #Y, D0
MOVE.W        D0, M
MOVE.W        A+2*3, M        {M:=A[3]. 2 because Integer }
```

END;

The following is incorrect:

```
ASM
    MOVE.W    X+Y,M    {reads address 11}
    MOVE.W    X+N,M    {reads 4 bytes past where N is
stored }
    MOVE.W    A,D0
    ADD.W     #3*2,D0    {really a[0]+6}
    MOVE.W    D0,M
END
```

ASSEMBLER procedures and functions

If the ASM...END statement is the only statement in a procedure (or function) the procedure can be declared as an ASSEMBLER procedure. This removes the need for the BEGIN and END block. The Add3 function can now be written as:

Function Add3(X,Y,Z: Integer): Integer; ASSEMBLER;

```
ASM
    move.w    X,d0
    add.w     Y,d0
    add.w     Z,d0
    move.w    d0,@result
```

END;

As well as looking much cleaner this also has the advantage that no copies are made of value parameters that are accessed via pointers. This means that faster, smaller code is generated in this instance, but you must be careful not to modify the parameters that have been passed.

In the example below a string is passed to the procedure by value, and the address of the string would be passed to the procedure, as described above. Under normal circumstances (without the Assembler directive) a copy would be made on entry to the procedure and you would then use an LEA instruction to access it.

With the Assembler directive `no copy` is made and you use the `MOVE` instruction to retrieve the address as follows:

```
Type C_Str=Packed Array [0..255] of Char;  
Procedure PasToCStr(P: String;var C: C_Str); ASSEMBLER;  
ASM
```

```
    MOVE.L    P,A0 {Address of Pascal string }  
    MOVE.L    C,A1 {Address of the C String}  
    MOVEQ     #0,D0  
    MOVE.B    (A0)+,D0 {String length}  
    BRA       @L2  
@L1: MOVE.B    (A0)+,(A1)+  
@L2: DBRA     D0,@L1  
      CLR.B    (A1) {Terminate the C String }  
END;
```

Using External Assembly files

The `{ $\$$ L FileName}` compiler directive allows assembly code to be included into your Pascal code. The object format used is the standard one used on the Amiga. They have the extension `.O` as default. When 'making' a program (compile with update check), the `$\$$ L` file is also checked to see if it has been changed.

HighSpeed Pascal makes some restrictions on what can be done using object files due to the nature of the built in smart linker in the compiler and because the object file first gets stored in the internal unit format. These are as follows:

1. All references to procedures (and functions) in other assembly modules or in the Pascal source must use the `JSR`, `JMP`, `LEA` or `PEA` formats. `BSR` and `BCC` are only legal within a module, because the assembler fixes these references itself. Also each procedure used for cross module reference must be named in the Pascal source by an external declaration, simply to tell the compiler that another entry in the procedure table is needed.

3. 8 bit fixups are not allowed as in `Var1 (A0,D0.W)`.

4. The registers that may be used are `D0`, `D1`, `D2`, `A0` and `A1`. All other used, must be saved and restored by the routine.

5. All names are restricted to 16 characters.

Following is two ways of implementing a function `AddThing` declared as shown in following program:

```
Program test;
{$L MyAsm}
Function AddThing(Thing1: Integer; VAR Thing2: Integer):
Integer; EXTERNAL;
```

```
Var n,SideVar: Integer;
```

```
Begin
```

```
    n:=AddThing(7,n);
```

```
end.
```

This is the standard way of making `AddThing` using assembly language:

```
; appropriate Devpac 3 options:
```

```
    OPT ALINK,NOCASE
```

```

XDEF AddThing                ;Make AddThing public
AddThing: LINK A6,#-0         ;= 0 because no locals
    MOVE.L 8(A6),A0           ;Read ^Thing2
    MOVE.W 8+4(A6),D0         ;Get Thing1
    ADD.W D0,(A0)             ;Update Thing2
    MOVE.W D0,8+4+2(A6)       ;Return D0 to AddThing
    UNLK A6
    MOVE.L (SP)+,A0           ;Get return address
    ADD #4+2,SP               ;Remove Thing2 and Thing1
    JMP (A0)                  ;and return
```

Note that the Devpac `rsset` and `rs` directives can be used to make the `(a6)` references more readable.

This is the quick and dirty way of making assembly files, also showing the use of a global Pascal variable `SideVar`.

```

XDEF AddThing          ;In file MyAsm.S
AddThing:MOVE.L (SP)+,RetAddr
MOVE.L (SP)+,A0        ;Read ^Thing2
MOVE.W (SP)+,D0        ;Get Thing1
ADDQ.W #1,SideVar      ;Side effect
ADD.W D0,(A0)          ;Update Thing2
MOVE.W D0,(SP)         ;Return D0 to AddThing
MOVE.L RetAddr,A0      ;Get return address
JMP (A0)               ;and return
BSS
XDEF SideVar           ;Global data
RetAddr: DS.L 1        ;Local data

```

Inline

Small assembly code subroutines can be placed directly in the source by declaring an inline procedure (or function). They should be short because all the code gets inserted every time the procedure or function is called. The syntax is:

```

InlineDirective =
    "Inline" Constant { "," Constant } .

```

Each constant is a 16 bit value. Following is an implementation of function AddThing using an inline directive:

Program test;

```

Function AddThing(Thing1: Integer; VAR Thing2: Integer):
Integer;

```

```

INLINE $205F, { MOVE.L (SP)+,A0 ;Read ^Thing2}
    $301F, { MOVE.W (SP)+,D0 ;Get Thing1}
    $D150, { ADD.W D0,(A0) ;Update Thing2}
    $3E80, { MOVE.W D0,(SP) ;Return D0}

```

```

var m,n: Integer; begin
    n:=AddThing(7,m);
end.

```

This is the input to HiSoft's Devpac assembler:

```

MOVE.L (SP)+,A0    ;Read ^Thing2
MOVE.W (SP)+,D0    ;Get Thing1
ADD.W D0,(A0)      ;Update Thing2
MOVE.W D0,(SP)     ;Return D0 to AddThing

```


and this is the assembly listing from Devpac, used in making the inline procedure heading:

HiSoft GenAm 680x0 Macro Assembler v3.01 Dec 10 1991

Page 1

myasm.s

```
1 00.00000000
2 00.00000000 205F      move.l  (sp)+,a0
3 00.00000002 301F      move.w  (sp)+,d0
4 00.00000004 D150      add.w   d0,(a0)
5 00.00000006 3E80      move.w  d0,(sp)
6 00.00000008
```

Device Drivers

It is possible to define your own devices in HighSpeed Pascal. They are activated by **Reset**, **Rewrite** and **Append** calls when referenced by their logical name. A device is installed with a call to:

Device(Name: String; IOprocedure: Pointer; DevBuf: TDevBuf).

The **DevBuf** is declared as

```
VAR DevBuf: TDevBuf; { buffer for device to use }
```

TDevBuf is a private type of the **System** module.

The call **Device('MyDriver',@MyDriver,Devbuf)** installs the name along with the address if the IO handler. The IO procedure must have a heading like this:

```
Function IOproc(var F: FileRec): Integer;
```

The **DevBuf** buffer must be a global variable to prevent it 'disappearing' when the procedure/function in which it is declared terminates thus causing the next call to **reset** or **rewrite** to crash.

The IO procedures determines what action it is to take by looking at the flags to see why it is called. The function value returned is put into the `IOResult` variable, thus forcing a runtime error if `IO+` is used. The routine gets called by `Write`, `WriteLn`, `Page`, `Read`, `ReadLn`, `Close`, `Eof`, `SeekEof`, `Eoln` and `SeekEoln` whenever input or output must be performed. `fInpFlag` is set when the driver should read `fBufSize` bytes (characters) into the buffer `fBuffer^`. `fBufPos` is then set to zero, and `fBufEnd` is set to show how many bytes actually got read. Whenever zero bytes are read, `Eof` is true.

`fOutFlag` is set when the driver should write `fBufPos` bytes from the buffer `fBuffer^`. `fBufPos` is set to zero if the data is written. It is legal not to do anything if the buffer is not full, that is if `fBufPos` does not equal `fBufEnd`. Here is a skeleton driver:

```
Function MyDriver(var F: FileRec): Integer;
```

```
var
```

```
    P: Integer;
```

```
    Ch: char;
```

```
begin
```

```
    MyDriver:=0; {IOResult:=Ok}
```

```
    With F do begin
```

```
        if fInpFlag then begin
```

```
            fBufEnd:=0;
```

```
            repeat
```

```
                Ch:=ReadKey;
```

```
                fBuffer^[fBufEnd]:=Ch; Inc(fBufEnd);
```

```
            until (Ch=#10) or (Ch=#Z) or (fBufEnd=fBufSize);
```

```
        end else {Doing output}
```

```
            {if fBufPos<>fBufEnd then {Always or when full}
```

```
            begin
```

```
                for P:=0 to fBufPos-1 do WriteCh(fBuffer^[P]);
```

```
                fBufPos:=0;
```

```
            end;
```

```
        end;
```

```
end;
```

CLI vs Workbench

There are two program environments on the Amiga® - the windows & icon driven Workbench, and the CLI or Shell. HighSpeed Pascal runs under either interface although most of the example programs can only run from a command line environment, the difference being in the startup and exit code.

CLI Startup

When a program is run from the CLI it starts with register A0 containing the address of the command line, and D0 containing its length. The DOS handles returned by `Input` and `Output` can be used for I/O with the console device and to exit, the program should simply RTS.

Workbench Startup

When a program is run from the Workbench it has to wait firstly for a message, and on terminating it has to reply to the message (after doing a `Forbid` call) before RTSing. The DOS functions `Input` and `Output` will not return valid handles, so you will need to open a window to perform any console I/O.

The startup differences are detailed in Chapter 30 of the *Amiga ROM Kernel Manual: Libraries and Devices*, together with the assembly language source of the startup code used by C programs.

A skeleton version of this for assembly language programmers can be found in the file `misc/easystart.i` which is included by the `freemem2.s` example program. It should be included at the very front of your programs, and handles the message-passing to allow programs to be run from the Workbench. Of course to do this you will need to create an icon for your program, using `Create Icons?` on the editor Settings menu.

Index

@result 267

■ A

Abs function 55
absolute variables 19
Addr function 56
append procedure 48, 56
Arc procedure, Graph unit 57
ArcTan function 58
arithmetic operators 24
array constants 21
arrays 11, 17
ASM statement 268
assembler
 DC directive 271
 external files 273
 operators 268
ASSEMBLER directive 272
assembler procedures 39
assembler, error messages 245
assembler, inline 266
assembler, instructions 270
assign procedure 47, 59
AssignCrt procedure, Crt unit. 60
assignment statement 31

■ B

Bar procedure, Graph unit 60
Bar3D procedure, Graph unit 61
BlockRead procedure 50, 62
blocks 43
BlockWrite procedure 50, 63
Boolean 8
boolean operators 25

■ C

case statements 33
Char 9
ChDir procedure, DOS unit 63
Chr function 64
Circle procedure, Graph unit 64
ClearDevice procedure, Graph unit 65
ClearViewport procedure, Graph unit 66
close procedure 48, 66
CloseGraph procedure, Graph unit 67
ClrEos procedure, Crt unit 68
ClrScr procedure, Crt unit 69
comments 2
compiler directives 6
compound statements 33
Concat function 70
constant 5
Copy function 70
Cos function 71

■ D

Dec 8
Dec procedure 72
Delay procedure 72
Delete procedure 73
DelLine procedure, Crt unit 73
DetectGraph procedure, Graph unit 74
devices handlers 54
Devpac 273

Dispose procedure 75
double 10
DrawPoly procedure, Graph unit
76
dynamic variables 17

■ E

Ellipse procedure, Graph unit 78
enumerated types 9
EnvCount function, DOS unit 78
EnvStr function, DOS unit 79
Eof function 49, 53, 80
Eoln function 53, 81
erase procedure 48, 81
error handling 54
error messages, AmigaDOS 241
error messages, assembler 245
Exec procedure, DOS unit 82
Exit procedure 83
Exp function 83
expressions 22
extended 10
external procedures 39

■ F

false 5
FExpand function, DOS unit 84
FExpandLock function, DOS unit 85
filepos function 49, 85
files 14
filesize function 49, 86
FillChar procedure 87
FillEllipse procedure, Graph unit
88
FillPoly procedure, Graph unit 89
FindFirst procedure, DOS unit 90
FindNext procedure, DOS unit 92
FloodFill procedure, Graph unit 93

for statements 34
forward procedures 38
FreeImage procedure, Graph unit
93
FreeMem procedure 94
FSplit procedure, DOS unit 95
function
activation 42
function calls 29
functions 37

■ G

GetArcCoords procedure, Graph
unit 96
GetAspectRatio procedure, Graph
unit 97
GetBkColor function, Graph unit
98
GetColor function, Graph unit 99
GetDate procedure, DOS unit 99
GetDefaultPalette procedure,
Graph unit 100
GetDir procedure, DOS unit 101
GetDriverName procedure, Graph
unit 102
GetEnv function, DOS unit 102
GetFattr procedure, DOS unit 103
GetFillPattern procedure, Graph
unit 104
GetFillSettings procedure, Graph
unit 106
GetFTime procedure, DOS unit 107
GetGraphMode procedure, Graph
unit 108
GetImage procedure, Graph unit
109
GetLineSettings procedure, Graph
unit 110
GetMaxColor function, Graph unit
111

Dispose procedure 75
double 10
DrawPoly procedure, Graph unit
76
dynamic variables 17

■ E

Ellipse procedure, Graph unit 78
enumerated types 9
EnvCount function, DOS unit 78
EnvStr function, DOS unit 79
Eof function 49, 53, 80
Eoln function 53, 81
erase procedure 48, 81
error handling 54
error messages, AmigaDOS 241
error messages, assembler 245
Exec procedure, DOS unit 82
Exit procedure 83
Exp function 83
expressions 22
extended 10
external procedures 39

■ F

false 5
FExpand function, DOS unit 84
FExpandLock function, DOS unit 85
filepos function 49, 85
files 14
filesize function 49, 86
FillChar procedure 87
FillEllipse procedure, Graph unit
88
FillPoly procedure, Graph unit 89
FindFirst procedure, DOS unit 90
FindNext procedure, DOS unit 92
FloodFill procedure, Graph unit 93

for statements 34
forward procedures 38
FreeImage procedure, Graph unit
93
FreeMem procedure 94
FSplit procedure, DOS unit 95
function
activation 42
function calls 29
functions 37

■ G

GetArcCoords procedure, Graph
unit 96
GetAspectRatio procedure, Graph
unit 97
GetBkColor function, Graph unit
98
GetColor function, Graph unit 99
GetDate procedure, DOS unit 99
GetDefaultPalette procedure,
Graph unit 100
GetDir procedure, DOS unit 101
GetDriverName procedure, Graph
unit 102
GetEnv function, DOS unit 102
GetFattr procedure, DOS unit 103
GetFillPattern procedure, Graph
unit 104
GetFillSettings procedure, Graph
unit 106
GetFTime procedure, DOS unit 107
GetGraphMode procedure, Graph
unit 108
GetImage procedure, Graph unit
109
GetLineSettings procedure, Graph
unit 110
GetMaxColor function, Graph unit
111

- GetMaxMode function, Graph unit 113
- GetMaxX function, Graph unit 113
- GetMaxY function, Graph unit 115
- GetMem procedure 116
- GetModeName function, Graph unit 117
- GetModeRange procedure, Graph unit 118
- GetPalette procedure, Graph unit 118
- GetPaletteSize function, Graph unit 119
- GetPixel function, Graph unit 121
- GetTextSettings procedure, Graph unit 121
- GettFillSettings procedure, Graph unit 122
- GetTime procedure, DOS unit 124
- GetViewSettings procedure, Graph unit 124
- GetX function, Graph unit 126
- GetY function, Graph unit 127
- global variables 16
- goto statements 32
- GotoXY procedure, CRT unit 128
- GraphDefaults procedure, Graph unit 128
- GraphErrorMsg function, Graph unit 129
- GraphResult function, Graph unit 130

■ H

- Halt procedure 132
- Hi function 132
- HighVideo procedure, Crt unit 133
- HiWord function 133

■ I

- identifiers 3
- if statements 33
- ImageSize procedure, Graph unit 133
- Inc 8
- Inc procedure 134
- Include function 135
- InitGraph procedure, Graph unit 135
- initialisers 20
- inline assembler 266
- inline directive 275
- inline procedures 39
- input 46
- Insert procedure 137
- InsLine procedure, Crt unit 138
- instructions, assembler 270
- Int function 139
- Integer 8
- IOResult function 49, 139

■ K

- KeyPressed function, Crt unit 140

■ L

- Length function 140
- Line procedure, Graph unit 141
- LineRel procedure, Graph unit 142
- LineTo procedure, Graph unit 142
- Ln function 143
- Lo function 143
- local variables 16
- LockAlertWindow function, DOS unit 144
- logical operators 25
- LoWord function 145

■ M

MaxAvail function 145
maxint 5
maxlongint 5
MemAvail function 146
MkDir procedure, DOS unit 146
Move procedure 147
MoveRel procedure, Graph unit 148
MoveTo procedure, Graph unit 149

■ N

New procedure 149
NormVideo procedure, Crt unit 150
numbers 3

■ O

Odd function 151
Omit function 152
operators
 arithmetic 24
 logical 25
 relational 26
 set 26
operators, assembler 268
Ord 8
Ord function 152
Ord4 function 153
Ordinal type 7
output 46
OutText procedure, Graph unit 153
OutTextXY procedure, Graph unit 154

■ P

PackTime procedure, DOS unit 154
Page procedure 54, 156
ParamCount function, DOS unit 156
parameters 40

ParamStr function, DOS unit 157
pi 5
Pi constant 159
PieSlice procedure, Graph unit 159
pointers 15, 18
Pos function 160
Pred 8
Pred function 161
procedure calls 32
procedures 37
 activation 42
program 44
Ptr function 161
PutImage procedure, Graph unit 162
PutPixel procedure, Graph unit 162

■ R

Random function 163
Randomize procedure 165
RandSeed variable 165
read procedure 49, 51, 166
ReadKey function, Crt unit 167
Readln procedure 52, 167
real types 10
record constants 21
records 12, 18
Rectangle procedure, Graph unit 168
relational operators 26
rename procedure 48, 169
repeat statements 35
reserved words 3
reset procedure 47, 169
RestoreCrtMode procedure, Graph unit 170
rewrite procedure 48, 171
Rmdir procedure, DOS unit 172
Round function 173

■ S

scope 45

Sector procedure, Graph unit 173

seek procedure 49, 174

SeekEof function 53, 174

SeekEoln function 53, 175

set constants 21

set operators 26

SetActivePage procedure, Graph unit 176

SetAllPalette procedure, Graph unit 178

SetAspectRatio procedure, Graph unit 180

SetBkColor procedure, Graph unit 181

SetColor procedure, Graph unit 181

SetCustomMode procedure, Graph unit 182

SetDate procedure, DOS unit 184

SetDateTime procedure, DOS unit 185

SetFAttr procedure, DOS unit 186

SetFillPattern procedure, Graph unit 187

SetFillStyle procedure, Graph unit 188

SetFTime procedure, DOS unit 190

SetGraphMode procedure, Graph unit 191

SetLineStyle procedure, Graph unit 192

SetPalette procedure, Graph unit 193

SetRGBPalette procedure, Graph unit 194

sets 14, 29

SetTextBuf procedure 54

SetTextJustify procedure, Graph unit 195

SetTextStyle procedure, Graph unit 197

SetTime procedure, DOS unit 199

SetViewport procedure, Graph unit 200

SetVisualPage procedure, Graph unit 201

SetWriteMode procedure, Graph unit 202

Simple type 7

Sin function 203

single 10

SizeOf function 203

SPtr function 204

Sqr function 204

Sqrt function 205

statements 30

Str procedure 205

Strings 4, 15, 17

structured constants 20

Structured types 11

subrange types 9

Succ 8

Succ function 206

Swap function 206

SwapWord function 206

■ T

text files 51

TextBackground procedure, Crt unit 207

TextColor procedure, Crt unit 208

TextHeight procedure, Graph unit 208

TextWidth procedure, Graph unit 209

true 5

Trunc function 209

typecast 30

typecasts 18
typed constants 20
typed files 49
types 6

■ U

units 44
UnPackTime procedure 210
untyped files 49
untyped variable parameters 41
UpCase function 210

■ V

Val procedure 211
value parameters 40
variables 16
variable parameters 41

■ W

WhereX function, Crt unit 212
WhereY function, Crt unit 212
while statements 36
WindMax function, Crt unit 213
WindMin function, Crt unit 213
with statements 36
write procedure 50, 52, 214
writeln procedure 52, 215

Notes

*HighSpeed Pascal for
Amiga Computers*

ISBN 0 948517 51 4

HiSoft
High Quality Software